

digital

## Guide to DECthreads

**OpenVMS**  
**OpenVMS**



---

## December 1995

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Digital or an authorized sublicensor.

© Digital Equipment Corporation 1991, 1992, 1993, 1994, 1995. All rights reserved.

The following are trademarks of Digital Equipment Corporation: AlphaServer, AXP, DEC, DEC Ada, DECdirect, DECladebug, DECthreads, Digital, Digital UNIX, OpenVMS, VAX, VAX Ada, VAX MACRO, VMS, ULTRIX, and the DIGITAL logo.

The following are third-party trademarks:

IEEE is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc.

Microsoft, Win32, and Windows 95 are registered trademarks, and Windows NT is a trademark of Microsoft Corporation.

OSF/1 is a registered trademark of the Open Software Foundation, Inc.

POSIX is a registered trademark of the IEEE.

Internet is a registered trademark of Internet, Inc.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Ltd.

All other trademarks and registered trademarks are the property of their respective holders.

ZK6101

This document is available on CD-ROM.

This guide supersedes the Guide to  
DECthreads, printed March 1994.  
OpenVMS Alpha Version 7.0  
OpenVMS VAX Version 7.0

Revision/Update information:

Software Version:

Digital Equipment Corporation  
Maynard, Massachusetts



---

# Contents

<b>Preface</b> .....	xv
----------------------	----

## **Part I DECthreads Overview and Programming Guidelines**

### **1 Introduction to Multithreaded Programming**

1.1	Advantages of Using DECthreads .....	1-1
1.2	Overview of Threads .....	1-1
1.3	Thread Execution .....	1-4
1.4	Software Models for Multithreaded Programming .....	1-5
1.4.1	Boss/Worker Model .....	1-5
1.4.2	Work Crew Model .....	1-5
1.4.3	Pipelining Model .....	1-6
1.4.4	Combinations of Models .....	1-6
1.5	Potential Problems with Multithreaded Programming .....	1-7
1.6	DECthreads POSIX 1003.1c Routines Summary .....	1-7
1.6.1	POSIX 1003.1c Routines Not Available .....	1-10
1.7	tis Routines Summary .....	1-11

### **2 Thread Concepts and Operations**

2.1	Thread Operations .....	2-1
2.1.1	Starting a Thread .....	2-1
2.1.2	Terminating a Thread .....	2-1
2.1.3	Termination .....	2-2
2.1.4	Waiting for a Thread to Terminate .....	2-2
2.1.5	Deleting a Thread .....	2-3
2.2	Attributes Objects .....	2-3
2.2.1	Creating an Attributes Object .....	2-3
2.2.2	Deleting an Attributes Object .....	2-3
2.2.3	Thread Attributes .....	2-4
2.2.3.1	Inherit Scheduling Attribute .....	2-4
2.2.3.2	Scheduling Policy Attribute .....	2-4
2.2.3.3	Scheduling Parameters Attribute .....	2-5
2.2.3.4	Stacksize Attribute .....	2-6
2.2.3.5	Guardsize Attribute .....	2-6
2.2.4	Mutex Attributes .....	2-6
2.2.4.1	Mutex Type Attribute .....	2-6
2.2.5	Condition Variable Attributes .....	2-6
2.3	Synchronization Objects .....	2-6



2.3.1	Mutexes .....	2-7
2.3.1.1	Normal Mutex .....	2-7
2.3.1.2	Recursive Mutex .....	2-7
2.3.1.3	Errorcheck Mutex .....	2-8
2.3.1.4	Mutex Operations .....	2-8
2.3.2	Condition Variables .....	2-9
2.3.3	Other Synchronization Methods .....	2-13
2.4	One-Time Initialization .....	2-13
2.5	Thread-Specific Data .....	2-13
2.6	Thread Cancellation .....	2-14
2.7	Thread Scheduling .....	2-17

### 3 Programming with Threads

3.1	Design for Asynchronous Execution .....	3-1
3.1.1	Memory Synchronization .....	3-2
3.2	Threads and Libraries .....	3-3
3.2.1	Thread Reentrant .....	3-3
3.2.2	Thread-Safe .....	3-3
3.2.3	Unsafe .....	3-4
3.2.3.1	DECthreads Global Lock .....	3-4
3.3	General Threaded Programming Issues .....	3-4
3.3.1	Shared Memory .....	3-4
3.3.1.1	Static .....	3-5
3.3.1.2	Stack .....	3-5
3.3.1.3	Heap .....	3-5
3.4	Stack Management .....	3-6
3.4.1	Stack Overflow .....	3-6
3.4.2	Sizing the Stack .....	3-6
3.5	Scheduling .....	3-7
3.5.1	Priority Inversion .....	3-7
3.6	Using Synchronization Objects .....	3-7
3.6.1	Mutex or Condition Variable .....	3-7
3.6.2	Race Conditions .....	3-8
3.6.3	Deadlocks .....	3-8
3.6.4	Signaling a Condition Variable .....	3-9
3.7	DECthreads Error Reporting .....	3-9
3.8	Multiple Threads Libraries Use Not Supported .....	3-11

### 4 Writing Thread-Safe Libraries

4.1	Mutexes .....	4-2
4.2	Condition Variables .....	4-2
4.3	Thread-Specific Data .....	4-2
4.4	Readers/Writer Locks .....	4-2

### 5 Using the DECthreads Exception Package

5.1	Overview of Exceptions .....	5-1
5.1.1	Types of Exceptions .....	5-2
5.1.2	Terminating Exception Semantics .....	5-2
5.2	Exception Operations .....	5-3
5.2.1	Declaring and Initializing an Exception Object .....	5-3
5.2.2	Raising an Exception .....	5-3



5.2.3	Defining a Code Region to Catch Exceptions .....	5-4
5.2.4	Catching a Particular Exception .....	5-4
5.2.5	Catching All Exceptions .....	5-5
5.2.6	Reraising the Current Exception .....	5-5
5.2.7	Defining Epilogue Actions for a Block .....	5-6
5.2.8	Determining the Current Exception .....	5-6
5.2.9	Importing a System-Defined Error Status into the Program as an Exception .....	5-6
5.2.10	Exporting a System-Defined Error Status .....	5-7
5.2.11	Reporting an Exception .....	5-7
5.2.12	Determining Whether Two Exceptions Match .....	5-7
5.3	C Language Syntax .....	5-8
5.4	Rules and Conventions for Modular Use of Exceptions .....	5-10
5.5	DECthreads Exceptions and Definitions .....	5-11

## 6 DECthreads Example

6.1	Prime Number Search Example .....	6-1
-----	-----------------------------------	-----

## Part II POSIX 1003.1c (pthread) Routines Reference

pthread_atfork .....	pthread-3
pthread_attr_destroy .....	pthread-5
pthread_attr_getdetachstate .....	pthread-6
pthread_attr_getguardsize_np .....	pthread-8
pthread_attr_getinheritsched .....	pthread-10
pthread_attr_getschedparam .....	pthread-12
pthread_attr_getschedpolicy .....	pthread-14
pthread_attr_getstacksize .....	pthread-16
pthread_attr_init .....	pthread-18
pthread_attr_setdetachstate .....	pthread-20
pthread_attr_setguardsize_np .....	pthread-22
pthread_attr_setinheritsched .....	pthread-24
pthread_attr_setschedparam .....	pthread-26
pthread_attr_setschedpolicy .....	pthread-28
pthread_attr_setstacksize .....	pthread-30
pthread_cancel .....	pthread-32
pthread_cleanup_pop .....	pthread-34
pthread_cleanup_push .....	pthread-35
pthread_cond_broadcast .....	pthread-37
pthread_cond_destroy .....	pthread-39
pthread_cond_init .....	pthread-41
pthread_cond_signal .....	pthread-43
pthread_cond_signal_int_np .....	pthread-45
pthread_cond_timedwait .....	pthread-47
pthread_cond_wait .....	pthread-49
pthread_condattr_destroy .....	pthread-51
pthread_condattr_init .....	pthread-52
pthread_create .....	pthread-54
pthread_debug .....	pthread-57



pthread_debug_cmd .....	pthread-58
pthread_delay_np .....	pthread-60
pthread_detach .....	pthread-61
pthread_equal .....	pthread-63
pthread_exit .....	pthread-64
pthread_get_expiration_np .....	pthread-66
pthread_getschedparam .....	pthread-68
pthread_getsequence_np .....	pthread-70
pthread_getspecific .....	pthread-71
pthread_join .....	pthread-72
pthread_key_create .....	pthread-74
pthread_key_delete .....	pthread-76
pthread_kill .....	pthread-78
pthread_lock_global_np .....	pthread-80
pthread_mutex_destroy .....	pthread-82
pthread_mutex_init .....	pthread-84
pthread_mutex_lock .....	pthread-86
pthread_mutex_trylock .....	pthread-88
pthread_mutex_unlock .....	pthread-90
pthread_mutexattr_destroy .....	pthread-92
pthread_mutexattr_gettype_np .....	pthread-93
pthread_mutexattr_init .....	pthread-95
pthread_mutexattr_settype_np .....	pthread-97
pthread_once .....	pthread-99
pthread_self .....	pthread-101
pthread_setcancelstate .....	pthread-102
pthread_setcanceltype .....	pthread-104
pthread_setschedparam .....	pthread-106
pthread_setspecific .....	pthread-109
pthread_sigmask .....	pthread-111
pthread_testcancel .....	pthread-113
pthread_unlock_global_np .....	pthread-114
sched_yield .....	pthread-115

### Part III Digital Proprietary Interfaces: tis Routines Reference

tis_cond_broadcast .....	tis-3
tis_cond_destroy .....	tis-4
tis_cond_init .....	tis-6
tis_cond_signal .....	tis-8
tis_cond_wait .....	tis-9
tis_getspecific .....	tis-11
tis_key_create .....	tis-12
tis_key_delete .....	tis-14
tis_lock_global .....	tis-15
tis_mutex_destroy .....	tis-16
tis_mutex_init .....	tis-18



tis_mutex_lock .....	tis-20
tis_mutex_trylock .....	tis-21
tis_mutex_unlock .....	tis-22
tis_once .....	tis-23
tis_raise .....	tis-25
tis_read_lock .....	tis-26
tis_read_trylock .....	tis-27
tis_read_unlock .....	tis-29
tis_rwlock_destroy .....	tis-30
tis_rwlock_init .....	tis-31
tis_self .....	tis-32
tis_setcancelstate .....	tis-33
tis_setspecific .....	tis-35
tis_testcancel .....	tis-37
tis_unlock_global .....	tis-38
tis_write_lock .....	tis-39
tis_write_trylock .....	tis-40
tis_write_unlock .....	tis-42

## Part IV Appendices

### A Considerations for Digital UNIX Systems

A.1	Overview .....	A-1
A.2	Digital UNIX Systems .....	A-2
A.2.1	Including DECthreads Header Files .....	A-2
A.2.2	Compiling Multithreaded Applications Libraries .....	A-2
A.2.3	Compiling Applications That Use the Draft 4 POSIX 1003.4a Standard Interface .....	A-3
A.2.4	Linking Multithreaded Shared Libraries .....	A-3
A.2.5	Compiling Applications That Use the Thread-Independent Services (tis) Interface .....	A-3
A.2.6	Support for Real-Time Scheduling on Digital UNIX Systems .....	A-3
A.2.7	Thread Cancelability of System Services .....	A-4
A.3	Using Signals .....	A-4
A.3.1	Types of Signals .....	A-4
A.3.1.1	Nonterminating Signals .....	A-4
A.3.1.2	Terminating Signals .....	A-4
A.3.1.3	Asynchronous Signals .....	A-4
A.3.1.4	Synchronous Signals .....	A-5
A.3.2	POSIX sigwait Service .....	A-5
A.3.3	Signal Alternatives Using the sigwait Routine .....	A-5
A.4	Signals Reported as Exceptions .....	A-6
A.4.1	Synchronous Terminating Signals .....	A-6
A.5	Dynamic Activation .....	A-6



## **B Considerations for OpenVMS Systems**

B.1	Overview .....	B-1
B.2	Including DECthreads Header Files .....	B-2
B.3	Compiling OpenVMS Images .....	B-2
B.4	Compiling OpenVMS Images Using the POSIX 1003.4a Draft 4 Standard .....	B-2
B.5	Linking OpenVMS Images .....	B-3
B.6	Using DECthreads with Asynchronous System Trap (AST) Routines ....	B-3
B.7	Dynamic Activation .....	B-4
B.8	Declaring an OpenVMS Condition Handler .....	B-4
B.9	Thread Cancelability of System Services .....	B-4
B.10	Using OpenVMS Alpha 64-Bit Addressing .....	B-4
B.11	DECthreads Condition Values .....	B-5
B.12	Two-Level Scheduling on OpenVMS Alpha .....	B-5
B.12.1	DECthreads Virtual Processors .....	B-6
B.12.2	AST delivery .....	B-7
B.12.3	Blocking System Services .....	B-8
B.12.4	\$HIBER and \$WAKE .....	B-8
B.12.5	Event Flags .....	B-9
B.12.6	Interactions with OpenVMS .....	B-9
B.12.7	Image Exit .....	B-10
B.12.8	Sysgen Parameter .....	B-10
B.12.9	Process Control System Services and DCL Commands .....	B-10
B.12.9.1	Process-Level System Services .....	B-10
B.12.9.2	Kernel-Level System Services .....	B-10
B.12.9.3	DCL Commands .....	B-11

## **C Considerations for Windows NT and Windows 95 Systems**

C.1	Using DECthreads Routines on Windows NT and Windows 95 Systems .....	C-1
C.2	Compiling DECthreads Applications .....	C-1
C.3	Linking DECthreads Applications .....	C-2
C.4	File Naming Support for DECthreads Header Files .....	C-2
C.5	Win32/DECthreads API Interoperability .....	C-2
C.6	Restrictions for Win32 API Routines .....	C-3
C.7	Thread Cancelability of System Services .....	C-3
C.8	Unsupported DECthreads Interface Routines .....	C-3
C.9	Use Restrictions of cma_delay and pthread_delay_np Routines .....	C-4
C.10	Timing Issues .....	C-4

## **D Debugging Multithreaded Applications**

D.1	DECthreads Debugger .....	D-1
D.1.1	Interactive Debugging .....	D-2
D.1.2	Noninteractive Debugging .....	D-2
D.1.3	Running DECthreads in Metered Mode .....	D-3
D.1.4	DECthreads Debugger Commands .....	D-3
D.1.4.1	Broadcast Command .....	D-3
D.1.4.2	Conditions Command .....	D-4
D.1.4.3	Exit and Quit Commands .....	D-4
D.1.4.4	Help Command .....	D-5
D.1.4.5	Keys Command .....	D-5
D.1.4.6	Mutexes Command .....	D-5



D.1.4.7	Signal Command .....	D-6
D.1.4.8	Show Command .....	D-6
D.1.4.9	Stack Command .....	D-6
D.1.4.10	System Command .....	D-7
D.1.4.11	Threads Command .....	D-7
D.1.4.12	Tset Command .....	D-8
D.1.4.13	Versions Command .....	D-9
D.1.4.14	VM Command .....	D-9
D.2	Debugging Threads on Windows NT Systems .....	D-10

## **E Digital Proprietary Interface Routines: CMA**

E.1	Migrating from a CMA Interface to POSIX 1003.1c .....	E-1
E.1.1	CMA Handles .....	E-1
E.1.2	Interface Routine Mapping .....	E-2
E.1.3	New POSIX 1003.1c Routines .....	E-4
E.2	Atomic Queues .....	E-4
E.3	Scheduling Parameters .....	E-4
E.4	CMA Exceptions .....	E-5
E.5	CMA Alert Example .....	E-9
E.6	CMA Routine Descriptions .....	E-10
	cma_alert_disable_asynch .....	E-11
	cma_alert_disable_general .....	E-12
	cma_alert_enable_asynch .....	E-13
	cma_alert_enable_general .....	E-15
	cma_alert_restore .....	E-16
	cma_alert_test .....	E-17
	cma_attr_create .....	E-18
	cma_attr_delete .....	E-20
	cma_attr_get_guardsize .....	E-21
	cma_attr_get_inherit_sched .....	E-22
	cma_attr_get_mutex_kind .....	E-23
	cma_attr_get_priority .....	E-24
	cma_attr_get_sched .....	E-25
	cma_attr_get_stacksize .....	E-26
	cma_attr_set_guardsize .....	E-27
	cma_attr_set_inherit_sched .....	E-28
	cma_attr_set_mutex_kind .....	E-30
	cma_attr_set_priority .....	E-31
	cma_attr_set_sched .....	E-33
	cma_attr_set_stacksize .....	E-35
	cma_cond_broadcast .....	E-36
	cma_cond_create .....	E-37
	cma_cond_delete .....	E-39
	cma_cond_signal .....	E-40
	cma_cond_signal_int .....	E-41
	cma_cond_timed_wait .....	E-42
	cma_cond_wait .....	E-44
	cma_debug .....	E-46
	cma_debug_cmd .....	E-47



cma_delay .....	E-48
cma_handle_assign .....	E-49
cma_handle_equal .....	E-50
cma_init .....	E-51
cma_key_create .....	E-52
cma_key_get_context .....	E-54
cma_key_set_context .....	E-55
cma_lock_global .....	E-56
cma_mutex_create .....	E-57
cma_mutex_delete .....	E-58
cma_mutex_lock .....	E-59
cma_mutex_try_lock .....	E-60
cma_mutex_unlock .....	E-61
cma_once .....	E-62
cma_stack_check_limit_np .....	E-65
cma_thread_alert .....	E-67
cma_thread_bind_to_cpu .....	E-68
cma_thread_create .....	E-69
cma_thread_detach .....	E-71
cma_thread_exit_error .....	E-72
cma_thread_exit_normal .....	E-73
cma_thread_get_priority .....	E-74
cma_thread_get_sched .....	E-75
cma_thread_get_self .....	E-76
cma_thread_join .....	E-77
cma_thread_set_priority .....	E-79
cma_thread_set_sched .....	E-81
cma_time_get_expiration .....	E-83
cma_unlock_global .....	E-84
cma_yield .....	E-85

## F DECthreads Library Routines

cma_lib_attr_create .....	F-2
cma_lib_attr_delete .....	F-3
cma_lib_attr_get_queuesize .....	F-4
cma_lib_attr_set_queuesize .....	F-5
cma_lib_queue_create .....	F-6
cma_lib_queue_delete .....	F-7
cma_lib_queue_dequeue .....	F-8
cma_lib_queue_enqueue .....	F-9
cma_lib_queue_requeue .....	F-10
cma_lib_queue_try_dequeue .....	F-11
cma_lib_queue_try_enqueue .....	F-12
cma_lib_queue_try_enqueue_int .....	F-13
cma_lib_queue_try_requeue .....	F-14



## G POSIX 1003.4a (Draft 4) pthread Routines

G.1	Migrating from a 1003.4a Interface to POSIX 1003.1c .....	G-1
G.1.1	Error Status and Function Returns .....	G-1
G.1.2	Replaced or Renamed Routines .....	G-2
G.1.3	Routines with No Changes to Syntax .....	G-2
G.1.4	Routines with Prototype or Syntax Changes .....	G-3
G.1.5	New Routines .....	G-4
	pthread_attr_create .....	G-5
	pthread_attr_delete .....	G-7
	pthread_attr_getdetach_np .....	G-8
	pthread_attr_getguardsize_np .....	G-9
	pthread_attr_getinheritsched .....	G-10
	pthread_attr_getprio .....	G-11
	pthread_attr_getsched .....	G-12
	pthread_attr_getstacksize .....	G-13
	pthread_attr_setdetach_np .....	G-14
	pthread_attr_setguardsize_np .....	G-16
	pthread_attr_setinheritsched .....	G-17
	pthread_attr_setprio .....	G-19
	pthread_attr_setsched .....	G-21
	pthread_attr_setstacksize .....	G-23
	pthread_bind_to_cpu_np .....	G-25
	pthread_cancel .....	G-26
	pthread_cleanup_pop .....	G-28
	pthread_cleanup_push .....	G-29
	pthread_condattr_create .....	G-30
	pthread_condattr_delete .....	G-31
	pthread_cond_broadcast .....	G-32
	pthread_cond_destroy .....	G-33
	pthread_cond_init .....	G-34
	pthread_cond_signal .....	G-36
	pthread_cond_signal_int_np .....	G-37
	pthread_cond_sig_preempt_int_np .....	G-39
	pthread_cond_timedwait .....	G-41
	pthread_cond_wait .....	G-43
	pthread_create .....	G-45
	pthread_delay_np .....	G-47
	pthread_detach .....	G-48
	pthread_equal .....	G-49
	pthread_exit .....	G-50
	pthread_get_expiration_np .....	G-51
	pthread_getprio .....	G-52
	pthread_getscheduler .....	G-53
	pthread_getspecific .....	G-54
	pthread_join .....	G-55
	pthread_keycreate .....	G-57
	pthread_lock_global_np .....	G-59



pthread_mutexattr_create .....	G-60
pthread_mutexattr_delete .....	G-61
pthread_mutexattr_getkind_np .....	G-62
pthread_mutexattr_setkind_np .....	G-63
pthread_mutex_destroy .....	G-64
pthread_mutex_init .....	G-65
pthread_mutex_lock .....	G-67
pthread_mutex_trylock .....	G-68
pthread_mutex_unlock .....	G-69
pthread_once .....	G-70
pthread_self .....	G-72
pthread_setasynccancel .....	G-73
pthread_setcancel .....	G-75
pthread_setprio .....	G-77
pthread_setscheduler .....	G-79
pthread_setspecific .....	G-81
pthread_testcancel .....	G-82
pthread_unlock_global_np .....	G-83
pthread_yield .....	G-84

## Glossary

## Index

## Examples

2-1	pthread Cancel Example .....	2-16
6-1	C Program Example (Prime Number Search) .....	6-2
E-1	CMA Alert Example .....	E-9

## Figures

1-1	Single Threaded Process .....	1-2
1-2	Multithreaded Process .....	1-3
1-3	Thread State Transitions .....	1-5
1-4	Work Crew Model of Thread Operation .....	1-6
1-5	Pipelining Model of Thread Operation .....	1-6
2-1	Only One Thread Can Lock a Mutex .....	2-7
2-2	Thread A Waits on Condition Ready .....	2-10
2-3	Thread B Signals Condition Ready .....	2-11
2-4	Thread A Wakes and Proceeds .....	2-12
2-5	Flow with FIFO Scheduling .....	2-18
2-6	Flow with RR Scheduling .....	2-19
2-7	Flow with Default Scheduling .....	2-19
4-1	Readers/Writer Lock Behavior .....	4-4



## Tables

1	Telephone and Direct Mail Orders .....	xvii
2	Conventions .....	xviii
1-1	DECthreads POSIX 1003.1c Routines Summary .....	1-8
1-2	DECthreads Routines Summary .....	1-11
5-1	pthread Exceptions .....	5-11
A-1	DECthreads Header Files .....	A-2
A-2	Digital UNIX Shared Libraries for Multithreaded Programs .....	A-2
A-3	Synchronous Terminating Signals .....	A-6
B-1	DECthreads Header Files .....	B-2
B-2	DECthreads Images .....	B-3
B-3	DECthreads Condition Values .....	B-5
D-1	DECthreads Debugging Return Status .....	D-2
E-1	CMA Exceptions .....	E-5



1	Exposure to noise (dB)	1
2	Exposure to noise (dB)	2
3	Exposure to noise (dB)	3
4	Exposure to noise (dB)	4
5	Exposure to noise (dB)	5
6	Exposure to noise (dB)	6
7	Exposure to noise (dB)	7
8	Exposure to noise (dB)	8
9	Exposure to noise (dB)	9
10	Exposure to noise (dB)	10
11	Exposure to noise (dB)	11
12	Exposure to noise (dB)	12
13	Exposure to noise (dB)	13
14	Exposure to noise (dB)	14
15	Exposure to noise (dB)	15
16	Exposure to noise (dB)	16
17	Exposure to noise (dB)	17
18	Exposure to noise (dB)	18
19	Exposure to noise (dB)	19
20	Exposure to noise (dB)	20
21	Exposure to noise (dB)	21
22	Exposure to noise (dB)	22
23	Exposure to noise (dB)	23
24	Exposure to noise (dB)	24
25	Exposure to noise (dB)	25
26	Exposure to noise (dB)	26
27	Exposure to noise (dB)	27
28	Exposure to noise (dB)	28
29	Exposure to noise (dB)	29
30	Exposure to noise (dB)	30
31	Exposure to noise (dB)	31
32	Exposure to noise (dB)	32
33	Exposure to noise (dB)	33
34	Exposure to noise (dB)	34
35	Exposure to noise (dB)	35
36	Exposure to noise (dB)	36
37	Exposure to noise (dB)	37
38	Exposure to noise (dB)	38
39	Exposure to noise (dB)	39
40	Exposure to noise (dB)	40
41	Exposure to noise (dB)	41
42	Exposure to noise (dB)	42
43	Exposure to noise (dB)	43
44	Exposure to noise (dB)	44
45	Exposure to noise (dB)	45
46	Exposure to noise (dB)	46
47	Exposure to noise (dB)	47
48	Exposure to noise (dB)	48
49	Exposure to noise (dB)	49
50	Exposure to noise (dB)	50
51	Exposure to noise (dB)	51
52	Exposure to noise (dB)	52
53	Exposure to noise (dB)	53
54	Exposure to noise (dB)	54
55	Exposure to noise (dB)	55
56	Exposure to noise (dB)	56
57	Exposure to noise (dB)	57
58	Exposure to noise (dB)	58
59	Exposure to noise (dB)	59
60	Exposure to noise (dB)	60
61	Exposure to noise (dB)	61
62	Exposure to noise (dB)	62
63	Exposure to noise (dB)	63
64	Exposure to noise (dB)	64
65	Exposure to noise (dB)	65
66	Exposure to noise (dB)	66
67	Exposure to noise (dB)	67
68	Exposure to noise (dB)	68
69	Exposure to noise (dB)	69
70	Exposure to noise (dB)	70
71	Exposure to noise (dB)	71
72	Exposure to noise (dB)	72
73	Exposure to noise (dB)	73
74	Exposure to noise (dB)	74
75	Exposure to noise (dB)	75
76	Exposure to noise (dB)	76
77	Exposure to noise (dB)	77
78	Exposure to noise (dB)	78
79	Exposure to noise (dB)	79
80	Exposure to noise (dB)	80
81	Exposure to noise (dB)	81
82	Exposure to noise (dB)	82
83	Exposure to noise (dB)	83
84	Exposure to noise (dB)	84
85	Exposure to noise (dB)	85
86	Exposure to noise (dB)	86
87	Exposure to noise (dB)	87
88	Exposure to noise (dB)	88
89	Exposure to noise (dB)	89
90	Exposure to noise (dB)	90
91	Exposure to noise (dB)	91
92	Exposure to noise (dB)	92
93	Exposure to noise (dB)	93
94	Exposure to noise (dB)	94
95	Exposure to noise (dB)	95
96	Exposure to noise (dB)	96
97	Exposure to noise (dB)	97
98	Exposure to noise (dB)	98
99	Exposure to noise (dB)	99
100	Exposure to noise (dB)	100



---

## Preface

This guide defines the disciplines of multithreaded code, providing implementation guidelines and concepts for thread-safe and multithreaded processing environments. The guidelines are supported with reference sections describing DECthreads routines of Digital's Multithreading Run-Time Library.

DECthreads provides the POSIX 1003.1c standard (pthread) interface and a Digital proprietary (cma) interface. DECthreads also provides a tis interface to assist libraries in supporting both threaded and nonthreaded environments. The interface you select depends upon your goals and the anticipated environment for your application.

### Intended Audience

This guide is intended for system and application programmers who want to create a multithreaded program using DECthreads routines.

### New and Changed Features

The *Guide to DECthreads* is essentially a completely revised document. The primary interfaces to DECthreads in previous operating system releases were the CMA interface and the Draft 4 POSIX 1003.4a standard (subsequently renamed 1003.1c) interface. This guide reflects that the primary DECthreads interface is now the approved POSIX 1003.1c-1995 standard interface. The CMA, the Draft 4 POSIX standard interface, and the Library reference sections have been moved to Appendices. Additionally, this guide documents that the TIS interface, which supports creation of thread-safe APIs in both threaded and nonthreaded environments, is now provided.

### Document Structure

This guide consists of the following:

#### Part I

- Chapter 1 provides a brief overview of multithreaded programming.
- Chapter 2 discusses the concepts and techniques related to DECthreads.
- Chapter 3 describes thread disciplines and coding issues you may face when writing a multithreaded program.
- Chapter 4 addresses writing thread-safe libraries.
- Chapter 5 introduces and provides conventions for the modular use of the DECthreads exception package.
- Chapter 6 contains examples demonstrating how to call DECthreads routines from a C language program.



## Part II

- This part provides detailed reference information on each DECthreads POSIX 1003.1c standard (pthread) routine of the interface. The pthread routine descriptions appear in alphabetical order by routine name.

## Part III

- This part provides detailed reference information on each DECthreads (Digital proprietary) tis routine. Routine descriptions appear in alphabetical order by routine name.

## Part IV - Appendixes

- Appendix A discusses DECthreads issues specific to Digital UNIX systems.
- Appendix B discusses DECthreads issues and restrictions specific to OpenVMS systems.
- Appendix C discusses DECthreads issues specific to the Microsoft Win32 interfaces of Windows NT and Windows 95 systems.
- Appendix D presents information on how to debug threads.
- Appendix E provides detailed reference information on each DECthreads (Digital proprietary) cma routine. Routine descriptions appear in alphabetical order by routine name.
- Appendix F provides detailed reference information on each DECthreads (Digital proprietary) library routine. Routine descriptions appear in alphabetical order by routine name.
- Appendix G provides detailed reference information on each DECthreads POSIX 1003.4a, Draft 4 standard (pthread) routine. These routines will be retired in a future release. They are provided here as a reference to assist code migration to the final POSIX compliant routines. The pthread routine descriptions appear in alphabetical order by routine name.

## Glossary

- The Glossary contains definitions of terms used in this guide, listed alphabetically.

## Related Documents

See your system's documentation set for more information on that system. DECthreads is available on the following platforms:

- Digital UNIX 4.0 (formerly DEC OSF/1)
- OpenVMS Alpha Version 7.0
- OpenVMS VAX Version 7.0

Digital has changed the name of its UNIX<sup>®</sup> operating system from DEC OSF/1 to Digital UNIX. The new name reflects Digital's commitment to UNIX and its conformance to UNIX standards.

For a complete list and description of the books in the OpenVMS documentation set, see the *Overview of OpenVMS Documentation*.



The printed version of the OpenVMS documentation set is color coded to help specific audiences quickly find the books that meet their needs. This color coding is reinforced with the use of an icon on the spines of books identifying the applicable user (U), manager (M), and programmer (P) audiences.

Some books in the OpenVMS or Digital UNIX documentation set help meet the needs of several audiences. For example, the information in some system manager, system administrator, or user books is also used by programmers. Keep this in mind when searching for information on specific topics. The *Documentation Overview, Glossary, and Master Index* provides information on all of the books in the OpenVMS or Digital UNIX documentation set.

## Reader's Comments

Digital welcomes your comments on this or any other guide. You can send comments in the following ways:

- Internet electronic mail: [writer@dceidl.enet.dec.com](mailto:writer@dceidl.enet.dec.com)
- FAX: 603-881-0120 Attn: Run-Time Libraries Group, ZKO2-3/Q18

If you have suggestions for improving particular sections or find any errors, please indicate the title, order number, and section numbers. Digital also welcomes general comments.

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book or on its back cover.)
- The type and version of the operating system that you are using. For example, Digital UNIX Version 4.0 or OpenVMS Version 7.0.
- If known, the type of processor that is running the operating system software. For example, AlphaServer 2000.
- The section numbers and page numbers of the information on which you are commenting.

Please address technical questions to your local system vendor or to the appropriate Digital technical support office. Information provided with the software media explains how to send problem reports to Digital.

## How to Order Additional Documentation

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825). To order additional documentation or information, see Table 1.

**Table 1 Telephone and Direct Mail Orders**

Location	Call	Fax	Write
U.S.A.	DECdirect 800-DIGITAL 800-344-4825	800-234-2298	Digital Equipment Corporation P.O. Box CS2008 Nashua, NH 03061

(continued on next page)



**Table 1 (Cont.) Telephone and Direct Mail Orders**

Location	Call	Fax	Write
Canada	DECdirect 800-267-6215	613-592-1946	Digital Equipment of Canada, Ltd. Box 13000 100 Hertzberg Road Kanata, Ontario, Canada K2K2A6
Internal	DTN: 264-4446 603-884-4446	603-884-3960	U.S. Software Supply Business

## Conventions

In this guide, every use of OpenVMS means the OpenVMS operating system and every use of UNIX means the Digital UNIX operating system.

Table 2 shows the conventions used in this guide.

**Table 2 Conventions**

Convention	Description
%	A percent sign represents the C shell system prompt.
\$	A dollar sign represents the system prompt for the VMS DCL interface, and Bourne and Korn shells.
#	A number sign represents the superuser prompt.
cat(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, cat(1) indicates that you can find information on the cat command in Section 1 of the reference pages.
Ctrl/x	The key combination Ctrl/x indicates that you must press the key labeled Ctrl while you simultaneously press another key, for example, Ctrl/Y or Ctrl/Z.
monospaced text	This typeface indicates the name of a command, routine, service, exception, or file. This typeface is also used in interactive examples and other screen displays. This typeface in lowercase is also used to indicate keywords, functions, files, and code statements when referencing a C language syntax.
<b>monospaced text</b>	This bolded typeface represents user input in interactive examples in the hardcopy and online versions of this guide.
...	A horizontal ellipsis in a figure or example indicates that not all of the statements are shown.
.	A vertical ellipsis indicates the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
()	In format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.

(continued on next page)



**Table 2 (Cont.) Conventions**

Convention	Description
[]	In format descriptions, brackets indicate that whatever is enclosed is optional; you can select none, one, or all of the choices.
{}	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
<b>boldface text</b>	Boldface text represents the introduction of a new term. Boldface text is also used to show user input in Bookreader versions of the guide.
<i>italic text</i>	Italic text represents book titles, parameters, arguments, and information that can vary in system messages (for example, Internal error <i>number</i> ).
numbers	Unless otherwise noted, all numbers in the text are assumed to be decimal. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.
mouse	The term <i>mouse</i> refers to any pointing device, such as a mouse, a puck, or a stylus.







# Part I

---

## DECthreads Overview and Programming Guidelines

Part I contains chapters that provide an overview and concepts of DECthreads as well as defining programming disciplines and guidelines for writing a multithreaded program.



# Part I

## Declarative Overview and Programming Guidelines

This book is a practical guide to the declarative programming paradigm. It is intended for programmers who are interested in learning the declarative programming paradigm and its applications. The book is divided into two parts. Part I, Declarative Overview and Programming Guidelines, provides a high-level overview of the declarative programming paradigm and its applications. Part II, Declarative Programming in Practice, provides a detailed guide to the declarative programming paradigm and its applications.



---

# Introduction to Multithreaded Programming

This chapter introduces **multithreaded programming**, which is the division of a program into multiple threads that execute concurrently. It describes four software models that can be used as a basis for constructing multithreaded programming programs and applications. The chapter also introduces concepts and techniques that are defined in more detail in Chapter 2.

## 1.1 Advantages of Using DECthreads

Threads are used to improve the performance (throughput, computational speed, responsiveness—or some combination) of a program. Multiple threads are useful in a multiprocessor system where threads run concurrently on separate processors. Threads created using the DECthreads library are capable of utilizing multiprocessors if the operating system on that platform supports parallelism within a process. Multiple threads also improve program performance on single processor systems by permitting the overlap of input, output, or other slow operations with computational operations.

Threads are useful in driving slow devices such as disks, networks, terminals, and printers. A multithreaded program can perform other useful work while waiting for the device to produce its next event (such as the completion of a disk transfer or the receipt of a packet from the network).

It is also advantageous to use threads when constructing a user interface. Consider the typical arrangement of a window system. Each time the user invokes an action (by clicking on a mouse button for example), a separate thread can be used to implement the action. If the user invokes multiple actions, multiple threads perform the actions in parallel. (Note that the implementation of the window system can also use a thread to handle the interaction with the user, because the user is an example of a slow device.)

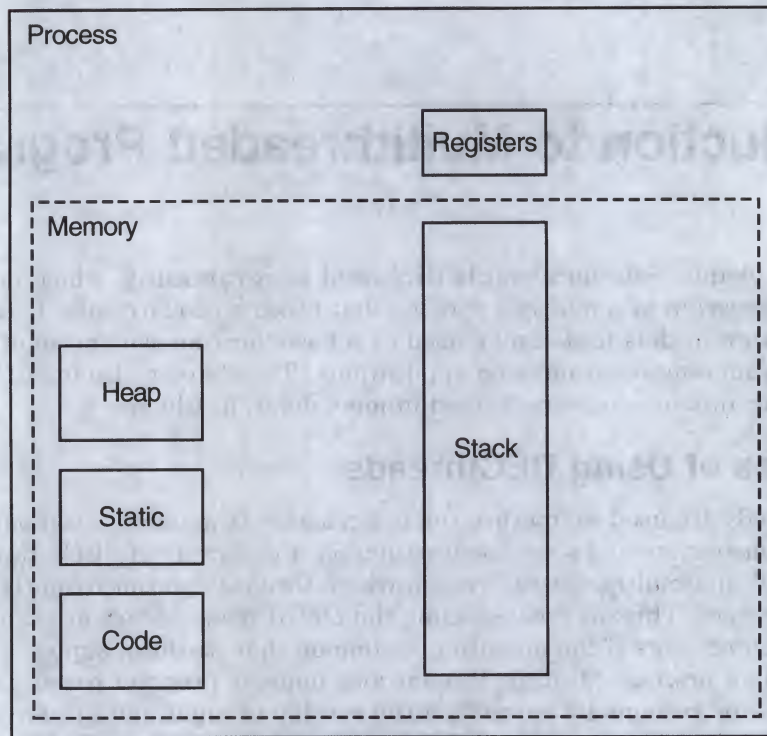
Threads are especially advantageous to use when building a distributed system. These systems frequently contain a shared network server, where the server services requests from multiple clients. Using multiple threads allows the server to handle clients' requests in parallel, instead of artificially serializing them (or creating one server process per client, at great expense).

## 1.2 Overview of Threads

A **thread** is a single, sequential flow of control within a program. Within each thread, there is a single point of execution. Most traditional programs consist of a single thread. Figure 1–1 and Figure 1–2 show the differences between a single threaded process and a multithreaded process.



**Figure 1-1 Single Threaded Process**

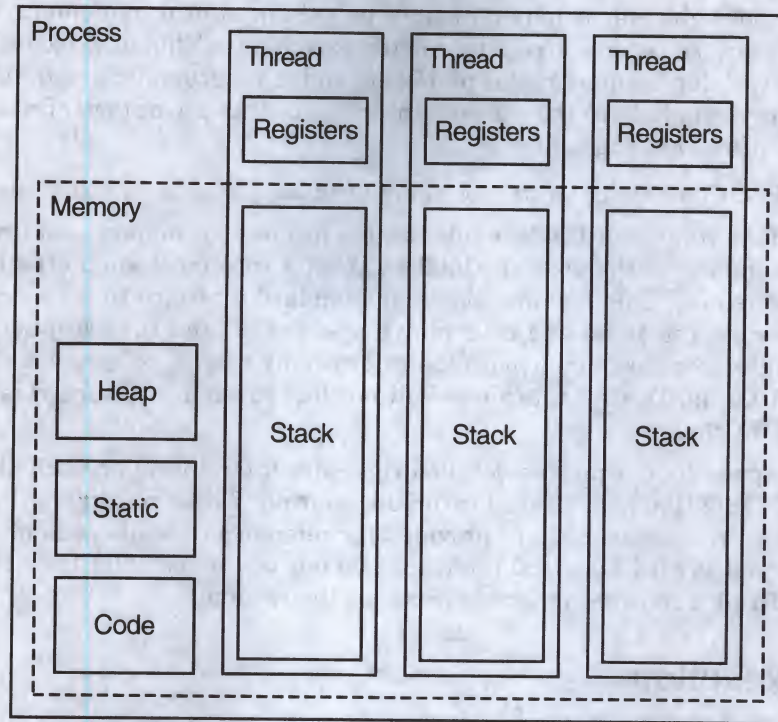


ZK-3913A-GE

Using DECthreads, Digital's multithreading run-time library, a programmer can create several threads within a program. Threads execute concurrently, and, within a multithreaded program, there are at any time multiple points of execution. Threads execute within (and share) a single address space. Therefore, threads read and write the same memory locations. Synchronization elements such as mutexes and condition variables ensure that the shared memory is accessed correctly. DECthreads provides routines that allow you to create and use these synchronization elements. Mutexes and condition variables are discussed in Section 2.3.1 and Section 2.3.2.



**Figure 1-2 Multithreaded Process**



ZK-3914A-GE

In Figure 1-2, notice that multiple threads share heap storage, static storage, and code but that each thread has its own register set and stack.

DECthreads provides the following four separate application programming interfaces (APIs) with which you can perform multithreaded operations:

- **pthread** interface

The **pthread** interface is an implementation of the IEEE Std 1003.1c-1995, POSIX Application Program Interface. More specifically, it is an extension to the 1003.1 Portable Operating System Interface (POSIX) standard rather than an independent interface specification. The DECthreads pthread implementation of the POSIX standard is the primary interface in the DECthreads environment. It is the most portable, efficient, and powerful interface in DECthreads.

The POSIX 1003.1c interfaces report errors by returning an integer value indicating the type of error.

- **tis** interface

The **tis** (thread independent services) is a Digital proprietary interface layered on top of the pthread interface. This API exists for building thread-safe code in nonthreaded applications. These routines provide their specified functionality when used in a program or application in which threads are present. In the absence of threads, these functions impose minimal overhead on the calling program. The tis objects created with these routines are DECthreads POSIX 1003.1c objects. You cannot create threads, since that has no meaning in a nonthreaded environment.

- **cma** interface



The **cma** interface is a Digital proprietary interface layered on top of the pthread interface. This interface to DECthreads will be obsolete in a future release in that it may no longer be enhanced or documented. The cma interface reports errors by raising exceptions. This interface is usually not available on non-Digital platforms and it is recommended that you migrate any cma code to the pthread interface to take advantage of new features and future enhancements.

- POSIX 1003.4a, Draft 4 obsolete interfaces

This version of DECthreads retains full binary support and limited source support for the obsolete 1003.4a Draft 4 interfaces supported by earlier versions. This includes both the standard interface that reports errors by setting `errno` and returning a value of -1, and the exception returning interface that, like cma, reports errors by raising exceptions. The POSIX 1003.4a, Draft 4 interfaces will not be provided in future versions of DECthreads.

Appendix G provides detailed reference information on each DECthreads POSIX 1003.4a, Draft 4 (pthread) routine. These routines will be retired in a future release, but are provided for reference to assist code migration to the final POSIX standard interface. Do not try to use 1003.1c or POSIX 1003.4a Draft 4 routines on cma objects, or the reverse.

## 1.3 Thread Execution

You can view multiple threads in a program as executing simultaneously. You cannot make any assumptions about the relative start or finish times of threads or the sequence in which they execute. Nevertheless, you can influence the scheduling of threads.

Each thread has its own thread identifier, which allows it to be uniquely identified. Also associated with a thread are its scheduling policy and priority, thread-specific data values, and the required system resources to support a flow of control.

A thread changes states during the course of its execution. A thread is in one of the following states:

- **Waiting**—The thread is not eligible to execute because it is synchronizing with another thread or with an external event, such as I/O.
- **Ready**—The thread is eligible to be executed by a processor.
- **Running**—The thread is currently being executed by a processor.
- **Terminated**—The thread has completed all of its work.

---

### Note

---

A multithreaded program must be reentrant. Therefore, be sure that your compiler generates reentrant code before you do multithreading design or coding work. (Digital's C, C++, Ada, Pascal, and BLISS compilers generate reentrant code by default.)

If your program is nonreentrant, it may be impossible to keep the program's threads from interfering with each other. See Section 3.2.1 for more information about thread reentrancy.

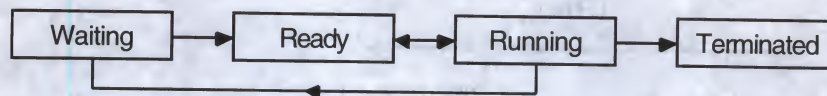
In general, when using threads, be aware of language common practices that are inherently not thread-safe. For example, FORTRAN typically



relies heavily upon static storage. These factors need to be addressed when writing threaded applications and thread-safe libraries.

Figure 1-3 shows the transitions between states for a typical thread implementation.

**Figure 1-3 Thread State Transitions**



ZK-3786A-GE

## 1.4 Software Models for Multithreaded Programming

The following sections describe four software models for which multithreaded programming is especially well suited:

- Boss/worker model
- Work crew model
- Pipelining model
- Combinations of models

### 1.4.1 Boss/Worker Model

In a boss/worker model of program design, one thread functions as the boss because it assigns tasks to worker threads for them to perform. Each worker performs a different task until it has finished, at which point it notifies the boss that it is ready to receive another task. Alternatively, the boss polls workers periodically to see whether or not each worker is ready to receive another task.

A variation of the boss/worker model is the work queue model. The boss places tasks in a queue, and workers check the queue and take tasks to perform. An example of the work queue model in an office environment is a secretarial typing pool. The office manager ("boss") puts documents to be typed in a basket and typists ("workers") take documents from the basket to work on.

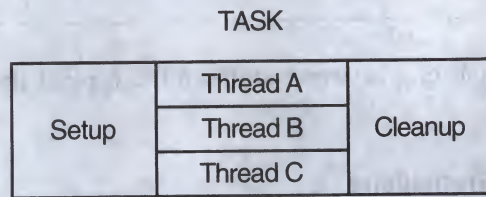
### 1.4.2 Work Crew Model

In the work crew model, multiple threads work together on a single task. The task is divided horizontally into pieces that are performed in parallel, and each thread performs one piece. An example of a work crew is a group of people cleaning a building. Each person cleans certain rooms or performs certain types of work (washing floors, polishing furniture, and so forth), and each works independently.

Figure 1-4 shows a task performed by three threads in a work crew model.



**Figure 1-4 Work Crew Model of Thread Operation**



(Time)



ZK-3787A-GE

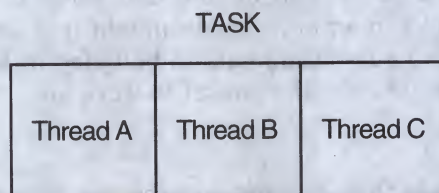
### 1.4.3 Pipelining Model

In the pipelining model, a task is divided vertically into steps. The steps must be performed in sequence to produce a single instance of the desired result, and the work done in each step (except for the first and last) is based on the previous step and is a prerequisite for the work in the next step. However, the program is designed to produce multiple instances of the desired result, and the steps are designed to operate in parallel so that while one step is performed on one instance of the result, the preceding step can be performed on the next instance of the result.

An example of the pipelining model in a factory environment is an automobile assembly line. Each step or stage in the assembly line is continually busy receiving the product of the previous stage's work, performing its assigned work, and passing the product along to the next stage.

In a multithreaded program using the pipelining model, each thread executes a step in the task. Figure 1-5 shows a task performed by three threads in a pipelining model.

**Figure 1-5 Pipelining Model of Thread Operation**



(Time)



ZK-3788A-GE

### 1.4.4 Combinations of Models

You may find it appropriate to combine the software models in a single program if your task is complex. For example, a program could be designed using the pipelining model, but with one or more steps handled by a work crew. In addition, threads could be assigned to a work crew by taking a task from a work queue and deciding (based on the task characteristics) which threads are needed for the work crew.



## 1.5 Potential Problems with Multithreaded Programming

When you design and code a multithreaded program, consider the following problems and accommodate or eliminate each as appropriate:

- **Program complexity**  
Program complexity is the most significant problem to consider in any multithreaded programming effort. Although using threads can simplify the coding and designing of a program, a certain level of expertise is required to be sure that the synchronization and interplay among threads is correct. This level of expertise is higher than for most single-threaded programs.
- **Race conditions**  
Race conditions from programming errors may cause unpredictable and erroneous program behavior. Section 3.6.2 discusses race conditions in more detail.
- **Deadlocks**  
Deadlocks from programming errors can cause two or more threads to be blocked from executing indefinitely. Section 3.6.3 discusses deadlocks in more detail.
- **Priority inversion**  
Priority inversion prevents high-priority threads from executing when interdependencies exist among three or more threads of different priorities. Section 3.5.1 discusses priority inversion in more detail.
- **Nonreentrant software**  
If a thread calls a routine or library that is not equipped to deal with threads, use the global locking mechanism to prevent conflicts with other threads using the same routine or library. Section 3.2.1 discusses nonreentrant software in more detail.

## 1.6 DECthreads POSIX 1003.1c Routines Summary

The highly portable pthread interface contains routines grouped in the following functional categories:

- General Threads Routines
- Attributes Object Routines
- Mutex Routines
- Condition Variable Routines
- Thread-Specific Data Routines
- Thread Cancellation Routines
- Thread Priority and Scheduling Routines

DECthreads also provides nonportable extensions in the following areas:

- Attributes Object Routines
- Debugging Support Routines
- Thread-Specific Data Routines



Table 1-1 lists and summarizes the DECthreads routines into these functional groups that support the POSIX thread (pthread) interface.

**Table 1-1 DECthreads POSIX 1003.1c Routines Summary**

Routine	Description
<b>General Threads Routines</b>	
pthread_atfork	Declares fork handlers to be called.
pthread_create	Creates a thread object and thread.
pthread_detach	Marks a thread object for deletion.
pthread_equal	Compares one thread identifier to another thread identifier.
pthread_exit	Terminates the calling thread.
pthread_join	Causes the calling thread to wait for the termination of a specified thread and detach it.
pthread_kill	Delivers a signal to a specified thread.
pthread_once	Calls an initialization routine to be executed only once.
pthread_self	Obtains the identifier of the current thread.
pthread_sigmask	Examines or changes the current thread's signal mask.
<b>Thread Attributes Object Routines</b>	
pthread_attr_destroy	Destroys a thread attributes object.
pthread_attr_getdetachstate	Obtains the detachstate attribute from the specified pthread attributes object.
pthread_attr_getinheritsched	Obtains the inherit scheduling attribute from the specified thread attributes object.
pthread_attr_getschedparam	Obtains the scheduling parameters for an attribute of the specified thread attributes object.
pthread_attr_getschedpolicy	Obtains the scheduling policy attribute of the specified thread attributes object.
pthread_attr_getstacksize	Obtains the stacksize attribute of a specified thread attributes object.
pthread_attr_init	Initializes a thread attributes object.
pthread_attr_setdetachstate	Changes the detachstate attribute in the specified thread attributes object.
pthread_attr_setinheritsched	Changes the inherit scheduling attribute of the specified thread attributes object.
pthread_attr_setschedparam	Changes the values of the parameters associated with the scheduling policy attribute of thread creation.
pthread_attr_setschedpolicy	Changes the scheduling policy attribute of the specified thread attributes object.
pthread_attr_setstacksize	Changes the stacksize attribute in the specified thread attributes object.

(continued on next page)



**Table 1-1 (Cont.) DECthreads POSIX 1003.1c Routines Summary**

Routine	Description
<b>Mutex Attributes Object Routines</b>	
pthread_mutexattr_init	Initializes a mutex attributes object.
pthread_mutexattr_destroy	Destroys a mutex attributes object.
<b>Condition Variable Attributes Object Routines</b>	
pthread_condattr_init	Initializes a condition variable attributes object that specifies condition variable attributes when created.
pthread_condattr_destroy	Destroys a condition variable attributes object.
<b>Mutex Routines</b>	
pthread_mutex_destroy	Destroys a mutex.
pthread_mutex_init	Initializes a mutex with attributes specified by the attributes argument.
pthread_mutex_lock	Locks an unlocked mutex. If locked, the caller waits for the mutex to become available.
pthread_mutex_trylock	Locks an unlocked mutex or returns immediately if mutex is locked.
pthread_mutex_unlock	Unlocks the mutex.
<b>Condition Variable Routines</b>	
pthread_cond_broadcast	Wakes all threads waiting on a condition variable.
pthread_cond_destroy	Destroys a condition variable.
pthread_cond_init	Initializes a condition variable.
pthread_cond_signal	Wakes at least one thread that's waiting on a condition variable.
pthread_cond_timedwait	Causes a thread to wait for a condition variable to be signaled or broadcasted for a specified period of time.
pthread_cond_wait	Causes a thread to wait for a condition variable to be signaled or broadcasted.
<b>Thread-Specific Data Routines</b>	
pthread_getspecific	Obtains the thread-specific data associated with the specified key.
pthread_key_create	Generates a unique thread-specific data key.
pthread_setspecific	Sets the thread-specific data value associated with the specified key for the current thread.
pthread_key_delete	Deletes a thread-specific data key.

(continued on next page)



**Table 1–1 (Cont.) DECthreads POSIX 1003.1c Routines Summary**

<b>Routine</b>	<b>Description</b>
<b>Thread Cancellation Routines</b>	
<code>pthread_cancel</code>	Allows a thread to request that it, or another thread terminate execution.
<code>pthread_cleanup_pop</code>	Removes a cleanup handler at the top of the cleanup stack and optionally executes it.
<code>pthread_cleanup_push</code>	Establishes a cleanup handler to be executed when the thread exits or is canceled.
<code>pthread_setcancelstate</code>	Sets the current thread's cancelability state.
<code>pthread_setcanceltype</code>	Sets the current thread's cancelability type.
<code>pthread_testcancel</code>	Requests delivery of any pending cancel to the current thread.
<b>Thread Priority and Scheduling Routines</b>	
<code>pthread_getschedparam</code>	Obtains the current scheduling policy and scheduling parameters of a thread.
<code>pthread_setschedparam</code>	Changes the current scheduling policy and scheduling parameters of a thread.
<code>sched_yield</code>	Notifies the scheduler that the current thread will release its processor to other threads of the same or higher priority.
<b>Non-Portable Extensions</b>	
Thread attribute routines	<code>pthread_attr_setguardsize_np</code> <code>pthread_attr_getguardsize</code>
Mutex attribute routines	<code>pthread_mutexattr_settype_np</code> <code>pthread_mutexattr_gettype_np</code>
Thread-specific data routines	<code>pthread_key_validate_np</code>
Debugging support routines	<code>pthread_debug</code> <code>pthread_debug_cmd</code> Static initialization of mutexes and condition variables with names.

### 1.6.1 POSIX 1003.1c Routines Not Available

The DECthreads POSIX 1003.1c Interface implementation does not support the following POSIX 1003.1c-1995 optionally implemented routines:

```

pthread_attr_getscope
pthread_attr_getstackaddr
pthread_attr_getstackaddr
pthread_attr_setscope
pthread_attr_setstackaddr
pthread_attr_setstackaddr
pthread_condattr_getpshared
pthread_condattr_setpshared
pthread_mutex_getprioceiling
pthread_mutex_setprioceiling
pthread_mutexattr_getprioceiling

```



```
pthread_mutexattr_getprotocol
pthread_mutexattr_getpshared
pthread_mutexattr_setprioceiling
pthread_mutexattr_setprotocol
pthread_mutexattr_setpshared
```

## 1.7 tis Routines Summary

Thread-independent services (**tis**) routines comprise a Digital proprietary interface of DECthreads. The **tis** interface provides services that assist with the development of thread-safe libraries. Thread synchronization can involve significant run-time cost, which is undesirable in a nonthreaded environment. In the nonthreaded environment, the **tis** interface enables you to build thread-safe libraries that are efficient, yet provide the necessary synchronization in the threaded environment. When DECthreads is not active within the process, **tis** executes only the minimum steps necessary. Code running in a nonthreaded environment is not burdened by the run-time synchronization that is necessary when the same code is run in a threaded environment. When DECthreads is active, the **tis** functions provide the necessary thread-safe synchronization.

In a nonthreaded environment, condition variables should not be used to block operations (for example, with **tis\_cond\_wait**). In a threaded environment, the guidelines for using the pthread routines apply to the use of the corresponding **tis** routine.

The **tis** routines can be classified into the following associated groups:

- General Routines
- Mutex Routines
- Condition Variable Routines
- Key Context Routines
- Cancellation Routines
- Readers/Writers Locks

Table 1–2 summarizes these groups of **tis** routines.

**Table 1–2 DECthreads Routines Summary**

Routine	Description
<b>General tis Routines</b>	
<b>tis_once</b>	Calls a one-time initialization routine that can be executed.
<b>tis_raise</b>	Sends a signal to the thread that called <b>tis_raise</b> .
<b>tis_self</b>	Obtains the identifier of the current thread.

(continued on next page)



**Table 1–2 (Cont.) DECthreads Routines Summary**

<b>Routine</b>	<b>Description</b>
<b>Mutex Routines</b>	
<code>tis_lock_global</code>	Locks a global <b>tis</b> mutex.
<code>tis_mutex_destroy</code>	Destroys a <b>tis</b> mutex.
<code>tis_mutex_init</code>	Initializes a <b>tis</b> mutex.
<code>tis_mutex_lock</code>	Locks an unlocked mutex.
<code>tis_mutex_trylock</code>	Tries to lock a mutex.
<code>tis_mutex_unlock</code>	Unlocks a <b>tis</b> mutex.
<code>tis_unlock_global</code>	Unlocks a global mutex.
<b>Condition Variable Routines</b>	
<code>tis_cond_broadcast</code>	Wakes all threads waiting on a condition variable.
<code>tis_cond_destroy</code>	Destroys a condition variable.
<code>tis_cond_init</code>	Initializes a condition variable.
<code>tis_cond_signal</code>	Wakes at least one thread that is waiting on a condition variable.
<code>tis_cond_wait</code>	Causes a thread to wait for a condition variable to be signaled or broadcasted.
<b>Key Context Routines</b>	
<code>tis_getspecific</code>	Obtains the data associated with the specified key.
<code>tis_key_create</code>	Generates a unique data key.
<code>tis_key_delete</code>	Deletes a data key.
<code>tis_setspecific</code>	Sets the data value associated with the specified key.
<b>Thread Cancellation Routines</b>	
<code>tis_setcancelstate</code>	Sets the current thread's cancelability state.
<code>tis_testcancel</code>	Creates a cancellation point in the current thread.
<b>Readers/Writers Locks</b>	
<code>tis_read_lock</code>	Acquires a read lock.
<code>tis_read_trylock</code>	Acquires a read lock; returns immediately if already locked.
<code>tis_read_unlock</code>	Unlocks a read lock.
<code>tis_rwlock_destroy</code>	Destroys a readers/writers lock.
<code>tis_rwlock_init</code>	Initializes a readers/writers lock.
<code>tis_write_lock</code>	Acquires a write lock.
<code>tis_write_trylock</code>	Acquires a write lock; returns immediately if already locked.
<code>tis_write_unlock</code>	Unlocks a write lock.



---

## Thread Concepts and Operations

This chapter discusses concepts and techniques related to DECthreads operations and coding. Chapter 1 summarized the operation of just the pthread routines. For detailed information on all the multithreading routines referred to in this chapter and the last, see the detailed description of the routines in the appropriate reference section.

---

### Note

The current version of DECthreads supports many similar API sets:

- POSIX 1003.1c-1995 interface
- TIS interface
- obsolete cma interface
- two variants of the obsolete POSIX 1003.4a, Draft 4 interface

The POSIX 1003.1c-1995 and tis interfaces are the primary DECthreads interfaces. Most functions have close equivalents in all of these API sets. Refer to the relevant appendices of this book for detailed comparisons.

---

## 2.1 Thread Operations

The following sections describe the operations you can perform with threads.

### 2.1.1 Starting a Thread

To start a thread, you can create it using the `pthread_create` routine. These routines create the thread object, based on the specified or default attributes, and start execution of the function you specified as the thread's start routine.

### 2.1.2 Terminating a Thread

A thread exists until it terminates and the thread has been detached. If the thread terminates before it is detached, then the thread continues to exist and other threads can join with it. Threads can be created in detached state, threads can be detached by calling `pthread_detach`, and are automatically detached when `pthread_join` returns.

---

### Note

When the initial thread returns from the main routine, the entire process (on UNIX systems) or image (on OpenVMS systems) terminates, just as it does when a thread calls `exit()` (on systems based on UNIX software) or `SYS$EXIT` (on OpenVMS systems).

---



A thread terminates for any of the following reasons:

- The thread returns from its start routine. This is the usual case.
- The thread calls the `pthread_exit` routine.

The `pthread_exit` routine terminates the calling thread and returns a status value (*value\_ptr* argument) to indicate the thread's exit status to any thread that calls `pthread_join`.

- The thread is terminated prematurely after having been specified in a call to the `pthread_cancel` routine.

The `pthread_cancel` routine requests termination of a specified thread if cancellation is permitted. See Section 2.6 for more information on canceling threads and controlling whether or not cancellation is permitted.

### 2.1.3 Termination

Termination occurs when a thread returns from its *start\_routine* function or calls `pthread_exit`. In the case of a termination, the following actions are performed:

1. The return value of the start routine function is copied into the thread object. This value can be obtained when another thread later calls the `pthread_join` routine. If the start routine returns normally and is a procedure that does not return a value, then the return value obtained by `pthread_join` will be unpredictable.
2. In the `pthread` interface to DECthreads, each cleanup handler that has been declared by `pthread_cleanup_push` and not yet removed by `pthread_cleanup_pop` is called. The most recently pushed handler is called first. Thread exit is accomplished by raising the exception `pthread_exit_e`. Using CATCH handlers in place of `pthread_cleanup_push` has the same effect, but is not portable.
3. Each thread-specific data destructor is removed from the list of destructors for this thread, and then is called. This step destroys all the thread-specific data associated with the current thread. See Section 2.5 for more information on thread-specific data.
4. Any thread currently waiting in a call to `pthread_join` for the terminating thread is awakened.
5. The thread object is marked to indicate that it is no longer needed by the thread itself. A check is made to determine if the thread is detached. If so, then the thread object is deallocated. Otherwise, the thread object is retained until it is detached.

### 2.1.4 Waiting for a Thread to Terminate

A thread waits for the termination of another thread by calling the `pthread_join` routine. Execution in the current thread is suspended until the specified thread terminates. Behavior is undefined if multiple threads call `pthread_join` and specify the same thread. This is because completion of the first join will detach the target thread.

If you specify the current thread with the `pthread_join` routine, a deadlock results. See Section 3.6.3 for more information about deadlocks.



Do not confuse `pthread_join` with other routines that cause waits and that are related to the use of a particular DECthreads feature. For example, use the `pthread_cond_wait` or `pthread_cond_timedwait` routines to wait for a condition variable to be signaled or broadcasted. (See Section 2.3.2 for more information on condition variables.)

### 2.1.5 Deleting a Thread

Once a thread is detached, it is automatically deleted after it terminates; that is, no explicit deletion operation is required. If the thread has not yet terminated, the `pthread_detach` routine marks the thread for deletion, and its storage is reclaimed immediately when the thread terminates. A thread cannot be joined or canceled after the `pthread_detach` routine has been called for the thread (even if the thread has not yet terminated).

If a thread that has not been detached terminates, its storage is retained so that other threads can join with it. The storage is reclaimed when the thread is detached.

## 2.2 Attributes Objects

An **attributes object** is used to describe DECthreads objects. This description consists of the individual attribute values that are used to create an object. An attributes object is analogous to a type definition in a programming language; it describes details of the objects to be created.

When you create an object, you can accept the default attributes for that object or specify an attributes object that contains specific attributes that you have set. For a thread, you can also change certain attributes after thread execution starts—for example, you can change the thread's priority.

The following sections describe how to create and delete attributes objects, and the individual attributes that you can specify for different objects.

### 2.2.1 Creating an Attributes Object

To create an attributes object, you can use one of the following routines, depending on the type of object to which the attributes apply:

- `pthread_attr_init` for thread attributes
- `pthread_condattr_init` for condition variable attributes
- `pthread_mutexattr_init` for mutex attributes

These routines create an attributes object containing default values for the individual attributes. To modify any attribute values in an attributes object, use one of the set routines described in the following sections.

Creating an attributes object or changing the values in an attributes object does not affect the attributes of objects previously created.

### 2.2.2 Deleting an Attributes Object

To delete an attributes object, use one of the following routines:

- `pthread_attr_destroy` for thread attributes objects
- `pthread_condattr_destroy` for condition variable attributes objects
- `pthread_mutexattr_destroy` for mutex attributes objects



Deleting an attributes object does not affect the attributes of objects previously created with that attributes object.

## 2.2.3 Thread Attributes

A **thread attributes object** allows you to specify values for thread attributes other than the defaults when you create a thread with the `pthread_create` routine. To use a thread attributes object, perform the following steps:

1. Create a thread attributes object by calling the `pthread_attr_init` routine.
2. Call the routines discussed in the following sections to set the individual attributes of the thread attributes object.
3. Create a new thread by calling the `pthread_create` routine and specifying the handle of the thread attributes object.

You have control over the following attributes of a new thread:

- Scheduling inheritance
- Scheduling policy
- Scheduling parameters
- Stack size
- Stack guard size

### 2.2.3.1 Inherit Scheduling Attribute

The **inherit scheduling attribute** specifies whether a newly created thread inherits the scheduling attributes (scheduling priority and parameters) of the creating thread (the default) or uses the scheduling attributes stored in the attributes object. You can set this attribute by calling the `pthread_attr_setinheritsched` routine.

### 2.2.3.2 Scheduling Policy Attribute

The **scheduling policy attribute** describes how the thread is scheduled for execution relative to the other threads in the program. A thread has one of the following scheduling policies:

- **SCHED\_FIFO (first-in/first-out (FIFO))**—The highest priority thread runs until it blocks. If there is more than one thread with the same priority and that priority is the highest among other threads, the first thread to begin running continues until it blocks. If a thread with this policy becomes ready, and it has a higher priority than the currently running thread, then it preempts the current thread and begins running immediately.
- **SCHED\_RR (round-robin (RR))**—The highest priority thread runs until it blocks; however, threads of equal priority, if that priority is the highest among other threads, are timesliced. (**Timeslicing** is a mechanism that ensures that every thread is allowed time to execute by preempting running threads at fixed intervals.) If a thread with this policy becomes ready, and it has a higher priority than the currently running thread, then it preempts the current thread and begins running immediately.
- **SCHED\_FG\_NP (also known as SCHED\_OTHER) (Default)**—All threads are timesliced. Under this policy, all threads receive some scheduling regardless of priority. Therefore, no thread is completely denied execution time. Nevertheless, higher priority threads receive more execution time than lower priority threads. Threads with the default scheduling policy can be denied execution time by FIFO or RR threads.



- **SCHED\_BG\_NP (Background)**—Like the default (throughput) scheduling policy, this policy ensures that all threads, regardless of priority, receive some scheduling. However, background threads can be denied execution time by FIFO or RR threads, and receive less execution time than default policy threads.

Note that only **SCHED\_FIFO** and **SCHED\_RR** are portable. The *name* **SCHED\_OTHER** is also portable, but its behavior may vary. For example, on some platforms it could be identical to **SCHED\_FIFO** or **SCHED\_RR**. The other policies are **DECthreads** extensions. You can use either of the following methods to set the scheduling policy attribute:

- Set the scheduling policy attribute in the attributes object, which establishes the scheduling policy of a new thread when it is created. To do this, call the `pthread_attr_setschedpolicy` routine. This allows the creator of a thread to establish the created thread's initial scheduling policy. Note that this value is used only if the attributes object is set so that the created thread does not inherit its priority from the creating thread. Inheriting priority is the default behavior.
- Change the scheduling policy of an existing thread (and, at the same time, the scheduling parameters) by calling the `pthread_setschedparam` routine. This has no effect on an attributes object.

Section 2.7 describes and shows the effect of the scheduling policy on thread scheduling.

### 2.2.3.3 Scheduling Parameters Attribute

The **scheduling parameters attribute** specifies the execution priority of a thread. (Although the terminology and format are designed to allow adding more scheduling parameters in the future, only priority is currently defined.) The priority is expressed relative to other threads in the same policy on a continuum of minimum to maximum for each scheduling policy. A thread's priority falls within one of the following ranges, depending on its scheduling policy.

Low	High
<b>PRI_FIFO_MIN</b>	<b>PRI_FIFO_MAX</b>
<b>PRI_RR_MIN</b>	<b>PRI_RR_MAX</b>
<b>PRI_OTHER_MIN</b>	<b>PRI_OTHER_MAX</b>
<b>PRI_FG_MIN_NP</b>	<b>PRI_FG_MAX_NP</b>
<b>PRI_BG_MIN_NP</b>	<b>PRI_BG_MAX_NP</b>

Section 2.7 describes how to specify priorities between the minimum and maximum values, and it also discusses how priority affects thread scheduling.

You can use either of the following methods to set the scheduling parameters attribute:

- Set the scheduling parameters attribute in the attributes object, which establishes the execution priority of a new thread when it is created. To do this, call the `pthread_attr_setschedparam` routine. This allows the creator of a thread to establish the created thread's initial execution priority. Note that this value is used only if the attributes object is set so that the created thread does not inherit its priority from the creating thread. Inheriting priority is the default behavior.



- Change the scheduling policy and parameters of an existing thread by calling the `pthread_setschedparam` routine. This allows a thread to change its own execution policy and/or priority.

#### 2.2.3.4 Stacksize Attribute

The **stacksize attribute** is the minimum size (in bytes) of the memory required for a thread's stack. To increase or decrease the size of the stack for the thread about to be created, call the `pthread_attr_setstacksize` routine, and use this attributes object when creating the thread and stack. You cannot change the size of a thread's stack after the thread has been created. See Section 3.4.2 for more information on sizing a stack.

#### 2.2.3.5 Guardsize Attribute

The **guardsize attribute** is the minimum size (in bytes) of the guard area for the stack of a thread. A **guard area** is a reserved area designed to help prevent or detect, or both, overflow of the thread's stack. The guard area is a region of memory that cannot be accessed by a thread. It is located adjacent to the last page in the thread's stack. To increase or decrease the size of the guard area for the thread about to be created, call the `pthread_attr_setguardsize_np` routine.

### 2.2.4 Mutex Attributes

A **mutex attributes object** allows you to specify values other than the defaults for mutex attributes when you initialize a mutex with the `pthread_mutex_init` routine.

Section 2.3.1 describes the purpose and types of mutexes.

#### 2.2.4.1 Mutex Type Attribute

The **mutex type attribute** specifies whether a mutex is normal, recursive, or errorcheck. (See Section 2.3.1 for more information.) You can set the mutex type attribute by calling the `pthread_mutexattr_settype_np` routine. If you do not use a mutex attributes object to select a mutex type, calling the `pthread_mutex_init` routine initializes a normal mutex by default.

### 2.2.5 Condition Variable Attributes

Currently, no attributes affecting condition variables are defined. You cannot change any attributes in the condition variable attributes object.

The `pthread_condattr_init` and `pthread_condattr_destroy` routines are provided for future expandability of the pthread interface, and to provide as much POSIX functionality as possible. These routines, in and of themselves, serve no useful function because there are no `pthread_condattr_set*` type routines available at this time.

Section 2.3.2 describes the purpose and uses of condition variables.

## 2.3 Synchronization Objects

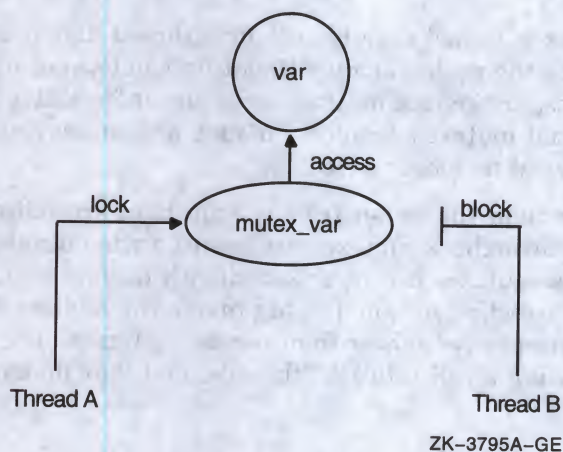
In a multithreaded program, you must use synchronization objects whenever there is a possibility of conflict in accessing shared data. The following sections discuss DECthreads synchronization objects: mutexes and condition variables.



### 2.3.1 Mutexes

A **mutex** (*mutual exclusion object*) is used by multiple threads to ensure the integrity of a shared resource that they access, most commonly shared data, by allowing only one thread to access it at a time. A mutex has two states, locked and unlocked. For each piece of shared data, all threads accessing that data must use the same mutex: each thread locks the mutex before it accesses the shared data and unlocks the mutex when it is finished accessing that data. If the mutex is locked by another thread, the thread requesting the lock either waits for the mutex to be unlocked or returns, depending on the lock routine called (see Figure 2-1).

Figure 2-1 Only One Thread Can Lock a Mutex



Each mutex must be initialized before use. DECthreads supports static initialization at compile time, using one of the macros provided in `<pthread.h>`, as well as dynamic initialization at run time by calling `pthread_mutex_init`. This routine allows you to specify an attributes object, which allows you to specify the mutex type. The types of mutexes are described in the following sections.

#### 2.3.1.1 Normal Mutex

A **normal mutex** (the default) is locked exactly once by a thread. If a thread tries to lock the mutex again without first unlocking it, the thread waits for itself to release the lock and deadlocks.

This is the most efficient form of mutex. When using interface and function inlining (optional), you can often lock and unlock a normal mutex without a call to DECthreads.

#### 2.3.1.2 Recursive Mutex

A **recursive mutex** can be locked more than once by a given thread without causing a deadlock. The thread must call the `pthread_mutex_unlock` routine the same number of times that it called the `pthread_mutex_lock` routine before another thread can lock the mutex. Recursive mutexes have the notion of a mutex owner. When a thread successfully locks a recursive mutex, it owns that mutex and the lock count is set to 1. Any other thread attempting to lock the mutex blocks until the mutex becomes unlocked. If the owner of the mutex attempts to lock the mutex again, the lock count is incremented, and the thread continues running. When an owner unlocks a recursive mutex, the lock count is decremented. The mutex remains locked and owned until the count reaches zero. It is an error for any thread other than the owner to attempt to unlock the mutex.



A recursive mutex is useful if a thread needs exclusive access to a piece of data, and it needs to call another routine (or itself) that needs exclusive access to the data. A recursive mutex allows nested attempts to lock the mutex to succeed rather than deadlock.

This type of mutex is called recursive because it allows you a capability not permitted by a normal (default) mutex. However, its use requires more careful programming. A recursive mutex should never be used with condition variables, because the unlock performed for a `pthread_cond_wait` or `pthread_cond_timedwait` might not actually release the mutex. In that case, no other thread can satisfy the condition of the predicate, and the thread waits indefinitely. See Section 2.3.2 for information on the condition variable wait and timed wait routines.

### 2.3.1.3 Errorcheck Mutex

An **errorcheck mutex** is locked exactly once by a thread, like a normal mutex. If a thread tries to lock the mutex again without first unlocking it, the thread receives an error. Thus, errorcheck mutexes are more informative than normal mutexes because normal mutexes deadlock in such a case, leaving you to determine why the thread no longer executes.

Also, if a thread other than the owner tries to unlock an errorcheck mutex, an error is returned. Errorcheck mutexes are useful during development and debugging. Errorcheck mutexes can be replaced with normal mutexes when the code is put into production use, or left to provide the additional checking. Errorcheck mutexes are always slower than normal mutexes. They cannot be locked without generating a call into DECthreads, and they do more internal tracking.

### 2.3.1.4 Mutex Operations

To lock a mutex, use one of the following routines, depending on what you want to happen if the mutex is locked:

- `pthread_mutex_lock`

If the mutex is locked, the thread waits for the mutex to become available.

- `pthread_mutex_trylock`

This routine returns immediately with a status indicating whether or not it was able to lock the mutex. Based on this return value, the calling thread can take the appropriate action.

When a thread is finished accessing a piece of shared data, it unlocks the associated mutex by calling the `pthread_mutex_unlock` routine. If other threads are waiting on the mutex, one is placed in the ready state. If more than one thread is waiting on the mutex, the scheduling policy (see Section 2.2.3.2) and the scheduling priority (see Section 2.2.3.3) determine which thread is readied, and the next running thread that requests it locks the mutex. The mutex is not automatically granted to the first waiter. If the unlocking thread attempts to relock the mutex before the first waiter gets a chance to run, the unlocking thread will succeed in relocking the mutex, and the first waiter may be forced to reblock.

You can delete a mutex and reclaim its storage by calling the `pthread_mutex_destroy` routine. Use these routines only after the mutex is no longer needed by any thread. A mutex cannot be deleted while it is locked.



---

### Important Note!

---

DECthreads does not currently detect deadlock conditions involving more than one mutex, but may in the future. **Never write code that depends upon DECthreads not reporting a particular error condition.**

---

## 2.3.2 Condition Variables

A **condition variable** allows a thread to block its own execution until some shared data reaches a particular state. A condition variable is a synchronization object used in conjunction with a mutex. A mutex controls access to shared data; a condition variable allows threads to wait for that data to enter a defined state. The state is defined by a Boolean expression called a **predicate**. A predicate may be a Boolean variable in the shared data or the predicate may be indirect; testing whether a counter has reached a certain value, or whether a queue is empty. Each predicate should have its own unique condition variable. Sharing a single condition variable between more than one predicate can introduce inefficiency or errors unless you use extreme care.

Cooperating threads test the predicate and wait on the condition variable if the predicate is not in the desired state. For example, one thread in a program produces work-to-do packets and another thread consumes these packets (does the work). If there are no work-to-do packets when the consumer thread checks, that thread waits on a work-to-do condition variable. When the producer thread produces a packet, it signals the work-to-do condition variable.

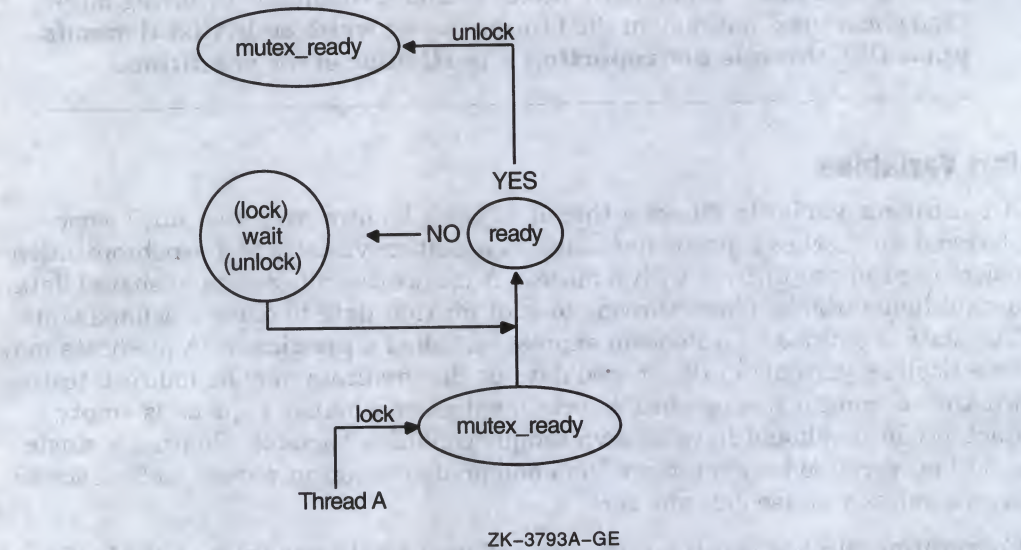
You must associate a mutex with a condition variable. A thread locks a mutex for some shared data and then tests the relevant predicate. If it is not in the proper state, the thread waits on a condition variable associated with the predicate. Waiting on the condition variable automatically unlocks the mutex. It is essential that the mutex be unlocked because another thread needs to acquire the mutex in order to put the data in the state required by the waiting thread. When the thread that acquires the mutex puts the data in the appropriate state, it wakes a waiting thread by signaling the condition variable. One thread comes out of its wait state with the mutex locked (the condition wait relocks the mutex before returning from the thread). Other threads waiting on the condition variable remain blocked.

It is important to wait on the condition variable and evaluate the predicate in a while loop. This ensures that the program will check the predicate after it returns from the condition wait. Note that threads are asynchronous. Another thread might consume the state before an awakened thread can run. Also, the test protects against spurious wakes that might happen and provides clearer program documentation.

For example, a Thread A may need to wait for a Thread B to finish a Task X before Thread A proceeds to execute a Task Y. Thread B can tell Thread A that it has finished Task X by putting a true or false value in a shared variable (the predicate). When Thread A is ready to execute Task Y, it looks at the shared variable to see if Thread B is finished (see Figure 2-2).



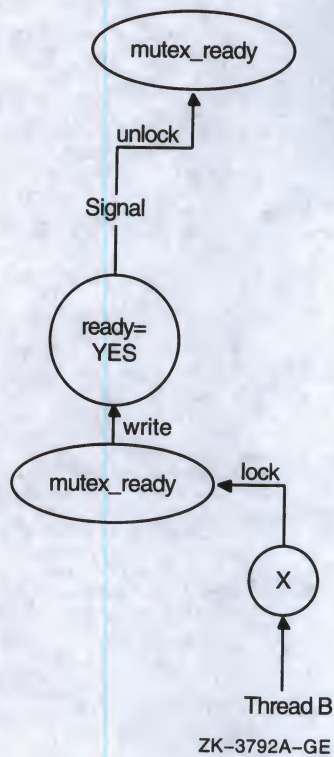
Figure 2-2 Thread A Waits on Condition Ready



First, Thread A locks the mutex named *mutex\_ready* that is associated with the shared variable named *ready*. Then it reads the value in *ready*. This test is called the predicate. If the predicate indicates that Thread B has finished Task X, then Thread A can unlock the mutex and proceed with Task Y. If the predicate indicates that Thread B has not yet finished Task X, however, then Thread A waits for the predicate to change by calling the `pthread_cond_wait` routine. This automatically unlocks the mutex, allowing Thread B to lock the mutex when it has finished Task X. Thread B updates the shared data (predicate) to the state Thread A is waiting for and signals the condition variable by calling the `pthread_cond_signal` routine (see Figure 2-3).



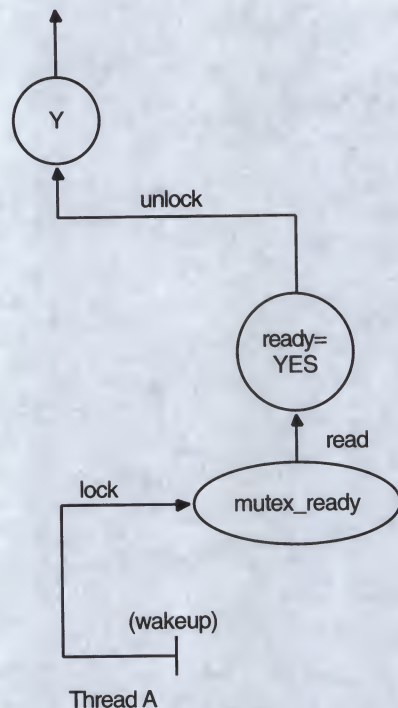
**Figure 2-3 Thread B Signals Condition Ready**



Thread B releases its lock on the shared variable's mutex. As a result of the signal, Thread A wakes up, implicitly regaining its lock on the condition variable's mutex. It then verifies that the predicate is in the correct state, and proceeds to execute Task Y (see Figure 2-4).



Figure 2-4 Thread A Wakes and Proceeds



ZK-3794A-GE

Note that although the condition variable is used for communication among threads, the communication is anonymous. Thread B does not necessarily know that Thread A is waiting on the condition variable that Thread B signals and Thread A does not know that it was Thread B that awakened it from its wait on the condition variable.

You can use the `pthread_cond_init` routine to initialize a condition variable. To create condition variables as part of your program's one-time initialization code, see Section 2.4. You can also statically initialize condition variables using one of the macros provided in `<pthread.h>`.

Use the `pthread_cond_wait` routine to cause a thread to wait until the condition is signaled or broadcasted. These routines specify a condition variable and a mutex that you have locked. (If you have not locked the mutex, the results of `pthread_cond_wait` are unpredictable.) These routines automatically unlock the mutex and cause the calling thread to wait on the condition variable until another thread calls one of the following routines:

- `pthread_cond_signal` to wake one thread that is waiting on the condition variable
- `pthread_cond_broadcast` to wake all threads that are waiting on a condition variable
- `pthread_cond_signal_int_np` to wake a thread from a signal handler (for UNIX) or AST routine (for OpenVMS). There are special restrictions on these functions (see Part II or Appendix E).



If a thread signals or broadcasts on a condition variable and there are no threads waiting at that time, the signal or broadcast has no effect. The next thread to wait on that condition variable blocks until the next signal or broadcast. (The `pthread_cond_signal_int_np` routine creates a pending wake condition, which causes the next wait on the condition variable to complete immediately.)

If you want to limit the time that a thread waits for a condition to be signaled or broadcasted, use the `pthread_cond_timedwait` routine. This routine specifies the condition variable, mutex, and absolute time at which the wait should expire if the condition variable has not been signaled or broadcasted.

You can destroy a condition variable and reclaim its storage by calling the `pthread_cond_destroy` routine. Use one of these routines only after the condition variable is no longer needed by any thread. A condition variable cannot be deleted while one or more threads are waiting on it.

### 2.3.3 Other Synchronization Methods

Another synchronization method that you can use is to call `pthread_join`. This routine allows a thread to wait for a specific thread to complete its execution. When the specified thread terminates, the joining thread is unblocked and continues its execution. See Section 2.1.2 for information on terminating a thread.

## 2.4 One-Time Initialization

You may have one or more routines that must be executed before any thread executes code in your facility, but that must be executed only once, regardless of the sequence in which threads start executing. For example, you may want to initialize mutexes, condition variables, or thread-specific data keys—each of which must be created only once—in an initialization routine. Multiple threads can call the `pthread_once` routine and DECthreads will ensure that the specified routine is called only once.

You can use the `pthread_once` routine to ensure that your initialization routine is executed only a single time, that is, by the first thread that tries to initialize the facility.

Alternately, you can statically initialize a mutex and a flag, and then simply lock the mutex and test the flag. In many cases, this is easier.

Finally, you may select initialization mechanisms, such as OpenVMS `LIB$INITIALIZE`, UNIX dynamic loader `__init_code`, or Win32 DLL initialization handlers.

## 2.5 Thread-Specific Data

Each thread has an area in which thread-specific data information is kept.

You can associate arbitrary data with a thread's context. You can think of this as the ability to add user-specified fields to the current thread's context or as global variables that have private values in each thread. A thread-specific data key is shared by all threads within the process—each thread has its own unique value for that shared key.

Use the following routines to create and access thread-specific data information:

- Use `pthread_key_create` to create a unique key value.



One call to the `pthread_key_create` routine creates a thread-specific data key shared by all threads. You can specify a destructor routine to destroy the context value associated with this key when any thread terminates. For example, to free heap storage the process must create each key exactly once—otherwise, subsequent creates will overwrite the first. See Section 2.4 for information about the one-time initialization in a threaded environment.

- Use `pthread_setspecific` to associate data with a key.

This routine associates some data with a specific key. Each thread can associate its own private data with the same key. For example, each thread might store a pointer to a block of dynamically allocated memory that it has reserved. Although each thread has its own block of memory, your code always uses the same key to get the current thread's block.

- Use `pthread_getspecific` to obtain the data associated with a key.

This routine obtains the current thread's thread-specific data value associated with a specified key.

## 2.6 Thread Cancellation

The **cancel** mechanism requests termination of another thread (or itself).

When you request that a thread be canceled, you are requesting that it terminate as soon as possible. However, the target thread can control how quickly it terminates by controlling its cancelability state and type.

A thread's initial cancelability state is enabled. **Cancelability state** determines whether a thread can receive a cancellation request. If the cancelability state is disabled, the thread does not receive any cancellation requests.

Initially, a thread's cancelability type is deferred. **Deferred** cancelability means that threads receive a cancellation request only at cancellation points—for example, when a call to the `pthread_cond_wait` routine is made. If you set a thread's cancelability type to asynchronous, the thread can receive a cancellation request at any time.

The following is a list of routines that are **cancellation points**:

- `pthread_setcanceltype` (when setting type to asynchronous)
- `pthread_testcancel`
- `pthread_delay_np`
- `pthread_join`
- `pthread_cond_wait`
- `pthread_cond_timedwait`

If the cancelability state is enabled, you can request the delivery of any pending cancel request by using the `pthread_testcancel` routine. This routine allows you to permit cancellation to occur at places where it might not otherwise be permitted, and it is especially useful within very long loops to ensure that cancel requests are noticed within a reasonable time.

If you set cancelability state to disabled, the thread cannot be terminated by any cancel request. This means that a thread could wait indefinitely if it does not come to a normal conclusion; therefore, exercise care.



When a cancellation request is delivered to a thread, the thread could be holding some resources, such as locked mutexes or allocated memory. Your program must release these resources before the thread terminates. DECthreads provides two equivalent mechanisms that can be used to do the cleanup during cancellation.

You can use the POSIX functions, `pthread_cleanup_push` and `pthread_cleanup_pop`, to establish and remove cleanup handlers for a section of code that contains a cancellation point. When a cancel request is delivered, the routine specified in `pthread_cleanup_push` is called. This allows the thread to unlock mutexes or otherwise release resources held in the current scope. Each routine can establish one or more cleanup handlers using `pthread_cleanup_push`. When the handler is no longer needed it is removed by calling `pthread_cleanup_pop`. The argument to `pthread_cleanup_pop` indicates whether the handler routine should be called when it is removed. Calling the cleanup handler automatically on removal is convenient when the thread is about to leave the scope and what you need is to perform the cleanup actions even though the thread wasn't cancelled (for example, releasing the mutex after waking up from a condition variable wait). (See also the pthread reference pages.)

Alternatively, you can use the DECthreads exception package TRY/CATCH or TRY/FINALLY macros to clean up during a cancellation request. A cancellation request is sent to the thread by raising a special DECthreads exception. Thus, code that contains a cancellation point can be placed inside a TRY block, and a CATCH or FINALLY block can be used to release the resources the thread is holding when the cancellation request is sent. Note that code should always reraise the cancellation exception—failing to do so will result in the thread not terminating as requested. (See also Chapter 5.)

Because it is impossible to predict exactly when an asynchronous cancellation request will be delivered, it is extremely difficult to recover properly when an asynchronous cancellation request is delivered. For this reason, an asynchronous cancelability type should only be set within regions of code that do not need to clean up in any way (such as unlocking mutexes or freeing storage). While cancelability type is asynchronous, do not call any routine unless it is explicitly documented as safe to be called with asynchronous cancelability type. Note that none of the general run-time routines and none of the DECthreads routines are safe except for `pthread_setcanceltype`.

For additional information on cancellation for your platform, see the appropriate section titled *Thread Cancelability of System Services* in Section A.2.7, Section B.9, and Section C.7.

---

#### Note

---

If the cancelability state is disabled, the thread cannot be canceled, regardless of the cancelability type. Setting cancelability type to asynchronous is relevant only when the thread's cancelability state is enabled.

---

Use the following routines to control the cancelability of threads:

- `pthread_setcancelstate` to set cancelability state
- `pthread_setcanceltype` to set cancelability type

A thread control and cancellation example is shown in Example 2-1.



### Example 2-1 pthread Cancel Example

```
/*
 * Pthread Cancel Example
 */

/*
 * Outermost cancellation state
 */
{
    .
    .
    .
    int s, outer_c_s, inner_c_s;
    .
    .
    .
    /* Disable cancellation, saving the previous setting. */
    s = pthread_setcancelstate (PTHREAD_CANCEL_DISABLE, &outer_c_s);
    if(s == EINVAL)
        printf("Invalid Argument!\n");
    else if(s == 0)
        .
        .
        /* Now cancellation is disabled. */
        .
        .
        /* Enable cancellation. */
        {
            .
            .
            s = pthread_setcancelstate (PTHREAD_CANCEL_ENABLE, &inner_c_s);
            if(s == 0)
                .
                .
                /* Now cancellation is enabled. */
                .
                .
                /* Enable asynchronous cancellation this time. */
                {
                    .
                    .
                    .
                    /* Enable asynchronous cancellation. */
                }
            .
            .
        }
    .
    .
}
```

(continued on next page)



### Example 2-1 (Cont.) pthread Cancel Example

```
int outerasync_c_s, innerasync_c_s;
.
.
s = pthread_setcanceltype (PTHREAD_CANCEL_ASYNCHRONOUS,
                           &outerasync_c_s);
if(s == 0)
.
.
/* Now asynchronous cancellation is enabled. */
.
.
/* Now restore the previous cancellation state (by
 * reinstating original asynchronous type cancel).
 */
s = pthread_setcanceltype (outerasync_c_s,
                           &innerasync_c_s);
if(s == 0)
.
.
/* Now asynchronous cancellation is disabled,
 * but synchronous cancellation is still enabled.
 */
}

.
.
}

.
.
/* Restore to original cancellation state. */
s = pthread_setcancelstate (outer_c_s, &inner_c_s);
if(s == 0)
.
.
/* The original (outermost) cancellation state is now reinstated. */
}
```

## 2.7 Thread Scheduling

Threads are scheduled according to their scheduling priority and how the scheduling policy treats those priorities. To understand the discussion in this section, you must understand the concepts in the following sections:

- Section 2.2.3.2 on scheduling policies, including how each policy handles thread scheduling priority
- Section 2.2.3.3 on thread scheduling priorities
- Section 2.2.3.1 on inheriting of scheduling attributes by created threads

To specify the minimum or maximum priority, use the appropriate symbol - for example, `PRI_OTHER_MIN` or `PRI_OTHER_MAX`. Priority values are integers, so you can specify a value between the minimum and maximum priority using an appropriate arithmetic expression. For example, to specify a priority midway between the minimum and maximum for the `SCHED_OTHER` scheduling policy, specify the following concept using your programming language's syntax:



$\text{pri\_other\_mid} = (\text{PRI\_OTHER\_MIN} + \text{PRI\_OTHER\_MAX}) / 2$

You should avoid using specific numerical values because the range of priorities can change from implementation to implementation. Values outside the range of minimum to maximum result in an error.

To show results of the different scheduling policies, consider the following example: A program has four threads, called A, B, C, and D. For each scheduling policy, three scheduling priorities have been defined: minimum, middle, and maximum. The threads have the following priorities:

A	minimum
B	middle
C	middle
D	maximum

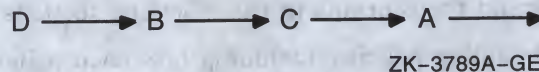
On a uniprocessor system, only one thread can run at any given time. The ordering of execution depends upon the relative scheduling policies and priorities of the threads. Given a set of threads with fixed priorities such as the previous list, their execution behavior is typically predictable. However, in a multiprocessor (SMP) system the execution behavior is much less determinable. Although the four threads have differing priorities, a 4-processor SMP system may execute all four simultaneously.

When you design an application to use thread priorities, it is critical to remember that scheduling is not the same as synchronization. You cannot assume that a high-priority thread can access shared data without interference from low-priority threads. Even if one of them is a FIFO policy thread at the highest priority and another is of background policy with lowest priority, they may run at the same time. On the other hand, you cannot even assume that on a 4-processor system, your four highest priority threads will be executing at any given time.

The following figures show uniprocessor execution flows depending on whether the first-in/first-out (FIFO), round-robin (RR), or throughput (Default) scheduling policy is in effect. Assume that all waiting threads are ready to execute when the current thread waits or terminates and that no higher priority thread is awakened while a thread is executing (during the flow shown in each figure).

Figure 2-5 shows a flow with FIFO scheduling.

**Figure 2-5 Flow with FIFO Scheduling**

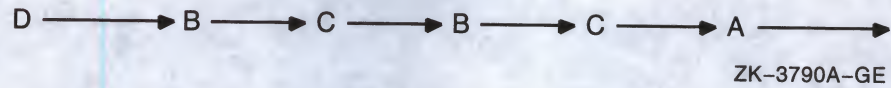


Thread D executes until it waits or terminates; then Thread B starts because it has been waiting longer than Thread C, and it executes until it waits or terminates; then Thread C executes until it waits or terminates; then Thread A executes.

Figure 2-6 shows a flow with RR scheduling.



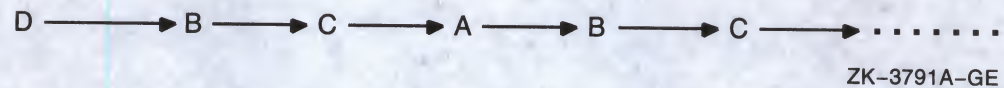
**Figure 2-6 Flow with RR Scheduling**



Thread D executes until it waits or terminates; then Threads B and C are timesliced, because they both have the same priority; then Thread A executes.

Figure 2-7 shows a flow with Default scheduling.

**Figure 2-7 Flow with Default Scheduling**



Threads D, B, C, and A are timesliced, even though Thread A has a lower priority than the others. Thread A receives less execution time than Thread D, B, or C if any of those are ready to execute as often as Thread A. However, the default scheduling policy protects Thread A against being blocked from executing indefinitely.

Because low-priority threads eventually run, the default scheduling policy protects against the problems of starvation and priority inversion, discussed in Section 3.5.1.



Figure 1. Flow chart of the study.

Figure 1. Flow chart of the study.

Figure 1. Flow chart of the study.

Figure 1. Flow chart of the study.

Figure 1. Flow chart of the study.

Figure 1. Flow chart of the study.

Figure 1. Flow chart of the study.



---

## Programming with Threads

This chapter discusses disciplines of which a programmer using threads must be aware. Pertinent examples include programming for asynchronous execution, choosing a synchronization mechanism, avoiding priority scheduling problems, making code thread-safe, and working with code that is not thread-safe.

### 3.1 Design for Asynchronous Execution

When programming with threads, always keep in mind that the execution of a thread is inherently asynchronous with respect to other threads running the system (or in the process). You *cannot* depend upon *any* synchronization between two threads unless you explicitly code that synchronization into your program using one of the following:

- Mutexes
- A properly tested application predicate loop on a condition variable
- A call to join with a thread you expect to terminate
- An equivalent platform dependent programming construct (such as VAX interlocked instructions; or Alpha load locked/store conditional sequences)

Some existing implementations of threads operate by context switching threads in user mode, within a single operating system process. Context switches between such threads occur only at relatively determinate times, such as when you make a blocking call to the threads library or when a timeslice interrupt occurs. This type of threading library might be termed “slightly asynchronous” because, with such a library, you can get away with many errors.

Systems that support kernel threads are less forgiving because context switches between threads can occur more frequently, and for less deterministic reasons. Systems that allow threads within a single process to run simultaneously on multiple processors are even less forgiving.

Some examples of common programming errors that may work often under some implementations but not at all under others are as follows:

1. Creating a thread with an argument that points to stack local data, or to global or static data that is serially reused for a sequence of threads.

There is no guarantee of when a thread will start. It can start immediately or not for a significant period of time, depending on the priority of the thread in relation to other threads that are currently running. When a thread will start can also depend on the behavior of other processes, as well as on other threaded subsystems within the current process.



Specifically, the thread started with a pointer to stack local data may not start until the creating thread's routine has returned, and the storage may have been changed by other calls. The thread started with a pointer to global or static data may not start until the storage has been reused to create another thread.

2. Initializing DECthreads objects (such as mutexes) or global data that is to be used by another thread *after* creating the thread.

On slightly asynchronous systems this is often safe because the thread will probably not run until the creator blocks. Thus, the error can go undetected initially. On another system (or in a later release of the operating system) that supports kernel threading, the created thread may run immediately, before the data has been initialized. This can lead to failures that are difficult to detect. Note that a thread may run to completion before the call that created it returns to the creator. The system load may affect the timing as well.

3. Using scheduling policy and priority as a synchronization mechanism.

In a uniprocessor system, only one thread can run at a time, and when a high-priority thread becomes runnable it immediately pre-empts a lower priority running thread. Therefore, a thread running at high priority might erroneously be presumed to not need a mutex to access shared data.

On a multiprocessor system, high and low-priority threads are likely to run at the same time. Situations can even arise where high-priority threads are waiting to run while the threads that are running have a lower priority.

Even if you know that your code is only going to run on a uniprocessor implementation, never try to use scheduling instead of synchronization. Your code will be safer, more portable, and upwardly compatible to a new release of the system with SMP support, if you design the code correctly in the first place.

Before you create a thread, you should set up all requirements that the thread will need to execute. If you need to set the thread scheduling parameters, for example, do so with attributes objects when you create it, rather than trying to use `pthread_setschedparam` or other routines afterwards. If you need to set global data for the thread or create synchronization objects, do these before you create the thread or set them in a `pthread_once` initialization routine that is called from each thread.

### 3.1.1 Memory Synchronization

Applications must ensure that access to data shared between threads is synchronized. POSIX 1003.1c-1995 guarantees that the following functions shall synchronize memory with respect to other threads:

`fork`, `pthread_create`, `pthread_join`, `pthread_mutex_lock`, `pthread_mutex_trylock`, `pthread_mutex_unlock`, `pthread_cond_wait`, `pthread_cond_timedwait`, `pthread_cond_signal`, `pthread_cond_broadcast`, `sem_post`, `sem_trywait`, `sem_wait`, `wait`, `waitpid`

Synchronization is not guaranteed if the function returns an error. For example, an unsuccessful `pthread_mutex_trylock` does not necessarily provide any synchronization.



## 3.2 Threads and Libraries

Because multithreaded programming has only recently become common, many existing code libraries are incompatible with threads. Many of the traditional C run-time library routines, for example, maintain state across multiple calls using static storage. This storage can become corrupted if routines are called from multiple threads at the same time. Even if the calls from multiple threads are serialized, code that depends upon a sequence of return values might not work. For example, the `getpwent(2)` routine returns the entries in the password file in sequence. If multiple threads call `getpwent` repeatedly, even if the calls are serialized, no thread will see all entries in the password file.

Library routines might be compatible with multithreaded programming to different extents. The important distinctions are thread safety and thread reentrancy.

### 3.2.1 Thread Reentrant

A routine is **thread-reentrant** when it functions normally despite being called simultaneously or sequentially by different threads. Again, taking the hypothetical example of `strtok`, the traditional interface could most efficiently be made thread-reentrant by adding an argument that specifies a context for the sequence of tokens. Thus, multiple threads could simultaneously parse different strings without interfering with each other.

The ideal thread-reentrant routine has no dependency on static data. Because static data must be synchronized using mutexes and condition variables, there is always a performance penalty due to the time required to lock and unlock the mutex and also in the loss of potential parallelism throughout the program. A routine that does not use any data that would be shared between threads can proceed without locking.

If you are developing new interfaces, make sure that any persistent context information (like the last-token-returned pointer in `strtok`) is passed explicitly so that multiple threads can process independent streams of information independently. Return information to the caller through routine values, output parameters (where the caller passes the address and length of a buffer), or by allocating dynamic memory and requiring the caller to free that memory when finished. Try to avoid using `errno` for returning error or diagnostic information; use routine return values instead.

### 3.2.2 Thread-Safe

A routine is called **thread-safe** if it can be called simultaneously from multiple threads without risk of corruption. Generally this means that it does some simple level of locking (possibly using the DECthreads global lock) to prevent simultaneously active calls in different threads. See Section 3.2.3.1 for information about the global lock.

Although these routines are thread-safe, they may be inefficient. For example, a UNIX `stdio` package that is thread-safe might still block all threads in the process while waiting to read or write data to a file. Routines such as `localtime(3)` or `strtok()`, which traditionally rely on static storage, could be made thread-safe by using thread-specific data instead of static variables.



### 3.2.3 Unsafe

When you must call code that is not thread-safe, you need to ensure serialization and exclusivity of the unsafe routine across all threads in the program. Using a mutex lock when calling any unsafe code accomplishes this. All threads and libraries using the routine should use the same mutex. Note that even if two libraries carefully lock a mutex around every call to a given routine, if each library uses a different mutex, the routine is not protected against multiple simultaneous calls from different libraries.

Furthermore, you must be aware that in many cases you need to protect more than just the call itself. You need to use or copy any static return values before releasing the mutex, and you may need to protect a sequence of calls rather than just a single call.

If a routine is not specifically documented as thread-reentrant or thread-safe, you should assume that it is unsafe. You should never assume that a routine is fully thread-reentrant unless that is specifically documented; many times, routines can rely on static data in ways that are not completely obvious from the interface. A routine carefully written to be thread-reentrant but that calls some other routine that is unsafe without proper protection, is actually unsafe itself.

#### 3.2.3.1 DECthreads Global Lock

DECthreads provides a single **global lock** that can be used by all threads in a program when calling routines or code that is not thread-safe to ensure serialization and exclusivity of the unsafe code. You can acquire the global lock by calling `pthread_lock_global_np` and unlock by calling `pthread_unlock_global_np`. The global lock allows a thread to acquire the lock recursively, so that you do not need to be concerned if you call a routine that also may acquire the global lock.

Because there is only one global lock, you do not need to fully analyze all of the dependencies in unsafe code that you call. With private locks to protect unsafe code, for example, one lock might protect calls to the `stdio` routine while another protects calls to math routines—but if `stdio` then calls a math routine without acquiring the math routine lock, the call is just as unsafe as if no locks were used.

Use the global lock whenever calling unsafe routines—and when unsure, always assume a routine is unsafe. All DECthreads routines are thread-safe.

## 3.3 General Threaded Programming Issues

This section addresses threaded programming disciplines concerning memory and stack issues.

### 3.3.1 Shared Memory

Most threads do not operate independently. They cooperate to accomplish a task, and cooperation requires communication. There are many ways that threads can communicate, and which method is most appropriate depends on the task. Threads that cooperate only rarely (for example, a boss thread that only sends off a request for workers to do long tasks) may be satisfied with a relatively slow form of communication. Threads that must cooperate more closely (for example, a set of threads performing a parallelized matrix operation) need fast communication—maybe even to the extent of using machine-specific atomic hardware operations.



Most mechanisms for thread communication involve the use of shared memory, taking advantage of the fact that all threads within a process share their full address space. Although all addresses are shared, there are three kinds of memory that are characteristically used for communication. The following sections describe the **scope** (the areas of the program where code can access the memory) and **lifetime** (the length of time the memory exists) of each of the three types of memory.

#### 3.3.1.1 Static

**Static memory** is allocated by the language compiler when it translates source code so the scope is controlled by the rules of the compiler. For example, in the C language, extern variables can be accessed anywhere, and static variables can be referenced within the source module or routine, depending on where they are declared. Note that the static memory described in this section is not the same as the C language static storage class; static memory refers to any variable that is permanently allocated at a particular address for the life of the program.

Static memory is appropriate when you know that only one instance of an object exists throughout the application. For example, if you want to keep a list of active contexts or a mutex to control some shared resource, you would not want individual threads to have their own copies of that data.

The scope of static memory depends on language scoping rules. The lifetime is the life of the program.

#### 3.3.1.2 Stack

**Stack memory** is allocated by code generated by the language compiler at run time, generally when a routine is initially called. When the program returns from the routine, the storage ceases to be valid (although the addresses still exist and may be accessible).

Generally, the storage is valid for the entire execution of the routine, and the actual address can be calculated and passed to other threads, but this depends on programming language rules. If you pass the address of stack memory to another thread, you must ensure that all other threads are finished processing that data before the routine returns; otherwise the stack will be cleared, and values may be altered by subsequent calls. The other threads will not be able to determine that this has happened and erroneous behavior will result.

The scope of stack memory is the routine or a block within the routine. The lifetime is no longer than the time during which the function executes.

#### 3.3.1.3 Heap

**Dynamic memory** is allocated by the program as a result of a call to some memory management function (for example, the C language run-time function `malloc()` or the OpenVMS common run-time function `LIB$GET_VM`).

Dynamic memory is referenced through pointer variables. Although the pointer variables are scoped depending on their declaration, the dynamic memory itself has no intrinsic scope or lifetime. It can be accessed from any routine or thread that is given its address and will exist until explicitly made free. In a language supporting automatic garbage collection, it will exist until the run-time system detects that there are no references to it. (If your language supports garbage collection, be sure the garbage collector is thread-safe.)



Heap is usually appropriate to manage persistent context. For example, in a thread-reentrant routine that is called multiple times to return a stream of information (for example, to list all active connections to a server or to return a list of users), using dynamic memory allows multiple contexts that are independent of threads. Multiple threads may be able to share a given context, or a single thread may have more than one context.

The scope of dynamic memory is anywhere a pointer containing the address can be referenced. The lifetime is from allocation to deallocation.

## 3.4 Stack Management

DECthreads sets a default stack size that is acceptable to most applications. You can also use the `stacksize` attribute to specify the stack size needed by any thread. This section discusses the cases in which the stack size is insufficient (resulting in stack overflow) and how to determine the optimal size of the stack.

Most VAX compilers do not probe the stack. Portable code that supports threads should use as little stack memory as practical. Most compilers on Alpha systems generate code in the procedure prologue to probe the stack ensuring there is enough space for the procedure to run. DECthreads provides **guard pages** (no access memory at the end of each thread's stack) to allow a very fast and simple probe. If you create a thread that might need to allocate large arrays on the stack, create the thread using an attributes object that specifies a large guard size attribute. A large stack guard region can help to prevent one thread from overflowing into another thread's stack region.

### 3.4.1 Stack Overflow

A program can receive a memory error (access violation, bus error, or segmentation fault) when it overflows its stack. It is often necessary to run the program under control of your system's debugger to determine where these errors occur. (However, if the debugger needs to allocate space on the stack, it may not function properly if the stack overflows.)

If a thread receives a memory access exception during a routine call or when accessing a local variable, increase the size of the stack. (To increase the thread's stack size attribute before creating it, call the `pthread_attr_stacksize` routine. See Section 2.2.3.4 for more information.) However, not all memory access exceptions indicate a stack overflow.

For programs that are not run under a debugger, determining a stack overflow is more difficult. This is especially true if the program continues to run after receiving a memory access exception. For example, if a stack overflow occurs while a mutex is locked, the mutex might not be released as the thread recovers or terminates. When the program attempts to lock that mutex again, it hangs.

### 3.4.2 Sizing the Stack

To determine the optimal size of a thread's stack, multiply the largest number of nested subroutine calls by the size of the call frames and local variables. Add to that number an extra amount of memory to accommodate interrupts. This process is difficult to perform because stack frames vary in size, and it might not be possible to estimate the depth of library function call frames.



You can also run your program using a profiling tool that measures actual stack use. This is commonly done by poisoning the stack before use by writing a distinctive pattern, and then checking for that pattern after the thread completes. DECThreads will use this mechanism when run with **metering** enabled. Note that any such monitoring tools will usually increase the amount of stack use.

## 3.5 Scheduling

Use care when writing code that uses realtime scheduling to control the priority of threads. First, review Section 3.1. Scheduling is not the same as synchronization. Second, note that threads with higher priority do not necessarily make your code run faster. Realtime priority adds overhead that can slow a program down, especially when interfacing with other libraries. For example, a high priority thread that polls for keyboard input may block work being done by other threads. Third, watch for pitfalls like priority inversion. It is best to avoid relying on realtime scheduling except where it's necessary to meet design goals. On the other hand, most systems that interact with external devices (including humans) have some realtime aspect.

### 3.5.1 Priority Inversion

**Priority inversion** occurs when interaction among three or more threads blocks the highest-priority thread from executing. For example, a high-priority thread waits for a resource locked by a low-priority thread, and the low-priority thread waits while a middle-priority thread executes. The high-priority thread is made to wait while a thread of lower priority (the middle-priority thread) executes.

To avoid priority inversion, associate a priority (at least as high as the highest priority thread that will use it) with each resource and force any thread using that object to first raise its priority to that associated with the object.

The Default (throughput) scheduling policy prevents priority inversion from causing a complete blockage of the high-priority thread, because the low-priority thread is eventually permitted to execute and release the resource. The FIFO and RR policies, however, do not provide for resumption of the low-priority thread if the middle-priority thread executes indefinitely.

## 3.6 Using Synchronization Objects

The following sections discuss when to use a mutex and when to use a condition variable and the use of mutexes to prevent two common problems: race conditions and deadlocks. This section also discusses why you should signal a condition variable with the associated mutex locked.

### 3.6.1 Mutex or Condition Variable

Use a mutex for tasks with fine granularity. Examples of fine-grained tasks are those that serialize access to shared memory or make simple modifications to shared memory (critical sections of a few program statements or less). Mutex waits are not interruptable—threads waiting to lock a mutex cannot be alerted or canceled.

A condition variable is not used to protect data. It is used to wait for data to assume a desired state. A condition variable is always used with a mutex that protects the shared data. Condition variable waits are interruptable.

See Section 2.3.1 and Section 2.3.2 for more information on mutexes and condition variables.



### 3.6.2 Race Conditions

A **race condition** occurs when two or more threads perform an operation, and the result of the operation depends on unpredictable timing factors; specifically, when each thread executes and waits and when each thread completes the operation.

For example, if two threads execute routines and each increments the same variable (such as  $X = X + 1$ ), the variable could be incremented twice and one of the threads could use the wrong value. For example:

1. Thread A increments variable X.
2. Thread A is interrupted (or blocked, or scheduled off) and Thread B is started.
3. Thread B starts and increments variable X.
4. Thread B is interrupted (or blocked, or scheduled off) and Thread A is started.
5. Thread A checks the value of X and takes some action based on that value.

The value of X is different from what it was when Thread A incremented it, and the program's behavior is incorrect.

Race conditions result from lack of (or ineffectual) synchronization. To avoid race conditions, ensure that any variable modified by more than one thread has only one mutex associated with it, and ensure that all accesses to the variable are made while holding that mutex.

See Section 3.6.4 for another example of a race condition.

### 3.6.3 Deadlocks

A **deadlock** occurs when a thread holding a resource is waiting for a resource held by another thread, while that thread is also waiting for the first thread's resource. Any number of threads can be involved in a deadlock if there is at least one resource per thread. A thread can deadlock on itself. Other threads can also become blocked waiting for resources involved in the deadlock.

Following are two techniques you can use to avoid deadlocks:

- Use sequence numbers with fast mutexes.

Associate a sequence number with each mutex and lock mutexes in sequence. Never attempt to lock a mutex with a sequence number lower than that of a mutex the thread already holds.

If a thread needs to acquire a mutex with a lower sequence number, it must first release all mutexes with a higher sequence number (after ensuring that the protected data is in a consistent state).

- Use a recursive mutex.

This method is useful when a thread needs to lock the same mutex more than once before unlocking it. This technique can help prevent a thread from deadlocking on itself.



### 3.6.4 Signaling a Condition Variable

When you are signaling a condition variable and that signal might cause the condition variable to be deleted, signal or broadcast the condition variable with the mutex locked.

The following C code fragment is executed by a releasing thread (Thread A):

```
pthread_mutex_lock (m);  
... /* Change shared variables to allow another thread to proceed */  
predicate = TRUE;  
pthread_mutex_unlock (m);  
pthread_cond_signal (cv);
```

①  
②

The following C code fragment is executed by a potentially blocking thread (Thread B):

```
pthread_mutex_lock (m);  
while (!predicate )  
    pthread_cond_wait (cv, m);  
pthread_mutex_unlock (m);  
pthread_cond_destroy (cv);
```

- ① If Thread B is allowed to run while Thread A is at this point, it finds the predicate true and continues without waiting on the condition variable. Thread B might then delete the condition variable with the `pthread_cond_destroy` routine before Thread A resumes execution.
- ② When Thread A executes this statement, the condition variable does not exist and the program fails.

The previous code fragments also demonstrate a race condition. The program depends on a sequence of events among multiple threads, but it does not enforce the desired sequence. Signaling the condition variable while holding the associated mutex eliminates the race condition. That prevents Thread B from deleting the condition variable until after Thread A has signaled it.

This problem can occur when the releasing thread is a worker thread and the waiting thread is a boss thread, and the last worker thread tells the boss thread to delete the variables that are being shared by boss and worker.

Code the signaling of a condition variable with the mutex locked as follows:

```
pthread_mutex_lock (m);  
... /* Change shared variables to allow some other thread to proceed */  
pthread_cond_signal (cv);  
pthread_mutex_unlock (m);
```

## 3.7 DECthreads Error Reporting

DECthreads can detect some of the following types of errors:

- Application programming interface (API) errors can occur when the program specifies an invalid parameter or attempts an inappropriate operation on some DECthreads object.
- Internal errors can occur when DECthreads determines that internal information has become corrupted to the point where it cannot continue operation.



API errors are reported in different ways by the various DECthreads interfaces:

- The POSIX 1003.1c (pthread) interface returns an integer value indicating the type of error.
- The cma interface raises exceptions to indicate error conditions.

DECthreads internal errors result in a bugcheck. DECthreads writes a message to the current error device (UNIX stderr or OpenVMS SYS\$ERROR) summarizing the problem, and creates a file containing more detailed information. By default, the file is named `pthread_dump.log` and is created in the current default directory. You can redirect the information to a different file by using the `PTHREAD_CONFIG dump` option.

If DECthreads cannot create the specified file when it performs the bugcheck, it will try to create the default file. If it cannot create the default file, it will write the detailed information to the error device.

The header message written to the error device starts with a line reporting that DECthreads has detected an internal problem and that it is terminating execution. It also includes the version of the DECthreads library. It will look something like the following:

```
%DECthreads bugcheck (version V3.13-180), terminating execution.
```

The subsequent line is the reason for the failure, and the final line written to the error device (usually) is the location of the detailed state information, as in the following example:

```
% Dumping to pthread_dump.log
```

The detailed information file contains information you can get from the `pthread_debug()` interface. This information is usually necessary to track down the problem. If you submit a problem report involving a DECthreads bugcheck, please include this information file along with sample code and output.

The fact that DECthreads terminated the process with a bugcheck can mean that some subtle problem in DECthreads has been uncovered. However, DECthreads does not check for all possible API errors, and there are a number of ways in which improper application code can result in a DECthreads bugcheck.

One common example is the use of any mutex operation or certain condition variable operations from within an interrupt routine (UNIX signal handler or OpenVMS AST routine). This type of programming error most commonly results in bugchecks reporting "enter\_kernel: deadlock" or "Can't find null thread". To prevent this error, avoid using any condition variables operations other than `pthread_cond_signal_int_np` from an interrupt routine (or the equivalent routines in other APIs).

In addition, DECthreads maintains a variety of state information in memory which is writable by user mode code. Therefore, it is possible for applications to accidentally modify DECthreads state by writing through invalid pointers, which can result in a bugcheck or other undesirable behavior.



### **3.8 Multiple Threads Libraries Use Not Supported**

DECthreads performs user mode execution "context switching" within a process or virtual processor by exchanging register sets (including the program counter and stack pointer). If any other code within the process also performs this sort of context switch, neither DECthreads nor that other code can ever know which context is active at any time. This can result in, at best, unpredictable behavior - and, at worst, severe errors. On OpenVMS VAX, for example, the VAX Ada run-time library provides its own specialized "tasking" package that does not use DECthreads scheduling. Therefore, the VAX Ada tasking cannot be used within a process that also uses DECthreads. (This restriction does not exist with DEC Ada for OpenVMS Alpha, as it uses DECthreads.)







---

## Writing Thread-Safe Libraries

This chapter addresses writing thread-safe libraries. Typically, these libraries do not use threads themselves. However, they need to be thread-safe because they may be called by applications that do use threads.

DECthreads provides the **tis** library to help you write efficient thread-safe code. The **tis** library provides thread-independent synchronization services that are very efficient when your code runs in a nonthreaded program (avoiding interlocked instructions and memory barriers), and provides full DECthreads synchronization in a threaded program.

It is not difficult to create thread-safe code and typically easy to convert existing code from unsafe to thread-safe, if the source code is available. The first consideration is whether the language compiler used in translating the source code supports reentrant code.

Most Ada compilers generate inherently reentrant code because Ada supports multithreaded programming. Although the C, Pascal, and BLISS languages do not support multithreaded programming directly, compilers for those languages generally create reentrant code. However, the Fortran and COBOL languages are defined in such a way that compilers may make implicit use of static storage, and such compilers do not generate reentrant code. It is difficult to write reentrant code in a nonreentrant language.

**tis** is designed to be fast when used without threads. For example, locking a mutex is essentially just setting a bit, and unlocking the mutex is just clearing the bit. However, all **tis** operations require a call into the **tis** library. That's because when DECthreads is initialized most of the **tis** functions are revectored so that subsequent calls use the normal DECthreads SMP-safe operations.

Once revectored, a `tis_mutex_lock()` call will function exactly as if `pthread_mutex_lock()` had been called. The transition from **tis** stubs to full DECthreads operation is transparent to library code using **tis**. If DECthreads is dynamically activated while a **tis** mutex is locked, the mutex can be unlocked normally.

The **tis** interface deliberately provides no way to determine whether DECthreads is active within the process. Thread-safe code should always act as if multiple threads may be active, because to do otherwise would inevitably result in problems (especially when you consider that DECthreads can be dynamically activated into the process at any time).

On Digital UNIX systems, **tis** is a part of `libc.so`. On OpenVMS systems, **tis** is implemented by `CMA$TIS_SHR.EXE`, which is included in `IMAGELIB.OLB` so the Linker can automatically resolve **tis** symbol references.



## 4.1 Mutexes

Just like normal DECthreads mutexes, tis mutexes provide synchronization between multiple threads that share resources. In fact, you can statically initialize tis mutexes using the POSIX 1003.1c `PTHREAD_MUTEX_INITIALIZER` macro defined in `pthread.h`, or any of the various non-portable DECthreads variants. That means you can create recursive or errorcheck mutexes if you need them, as well as the default normal mutexes. You can assign names to your tis mutexes.

Unlike static initialization, however, dynamic initialization of tis mutexes is limited by the absence of attributes objects. The `tis_mutex_init()` routine can create only **normal** (default) mutexes.

When DECthreads is initialized, any locked mutexes remain locked. The ownership of recursive and errorcheck mutexes remains valid. That means it's legal and reasonable to lock a tis mutex, dynamically activate the DECthreads library, and then unlock the mutex.

The DECthreads **global lock** is implemented in tis, and you can use it without calling the DECthreads interfaces by calling `tis_lock_global()` and `tis_unlock_global()`.

## 4.2 Condition Variables

As for mutexes, tis condition variables look just like DECthreads condition variables. You can statically initialize them using `PTHREAD_COND_INITIALIZER` or any of the non-portable DECthreads variants. You can assign names to your tis condition variables, for example, by statically initializing them with `PTHREAD_COND_INITWITHNAME()`.

Also as for mutexes, dynamic initialization of tis condition variables is limited by the absence of attributes objects.

Signalling or broadcasting a tis mutex when DECthreads isn't present does nothing. That's because you're not allowed to wait on a tis condition variable without DECthreads. Note that you can have more than one thread only if DECthreads is present; so if you waited, there would be no threads to wake you up.

## 4.3 Thread-Specific Data

When DECthreads is initialized, tis thread-specific data keys are transferred into DECthreads, so the same keys can continue to be used. Any thread-specific data values set using tis will be transferred into the default (initial) thread.

## 4.4 Readers/Writer Locks

Readers/writer locks exist only within the tis interface. There are no equivalent pthread or cma interfaces. That's ok, since you can use tis even if you build your program with DECthreads. Keep in mind that readers/writer locks are not currently portable—although the IEEE POSIX 1003.1j standard, currently in balloting, proposes a very similar set of functions that will eventually be made available in DECthreads.

Readers/writer locks routines can be used to control access to any shared resource and can be executed by a thread or program. Readers/writer locks are appropriate to protect information that needs to be read frequently and written only occasionally. For example, in a cache of recently accessed information, many



threads may simultaneously examine the cache without conflict. When a thread needs to update the cache, it must have exclusive access.

A readers/writer lock can be locked for a shared read access, or for exclusive write access. A read access lock will block when any thread or program has an active write access. A write access will block when another thread currently has either write access or read access.

When both readers and writers are waiting for access at the same time, the readers are given precedence when the write lock is released. Read precedence favors concurrency because it potentially allows many threads to accomplish work simultaneously. Figure 4-1 illustrates the lock behavior on three threads (one writer and two readers) competing for the same memory object.

The `tis_rwlock_init` initializes a readers/writer lock by allocating and initializing the `tis_rwlock_t` structure.

You call `tis_read_lock` or `tis_write_lock` when access to the shared resource is required. `tis_read_trylock` and `tis_write_trylock` can also be called to acquire a lock, but if the resource is already locked by another caller, the trylock routine immediately returns `EBUSY`, rather than waiting.

Call `tis_rwlock_destroy` when you are finished using a readers/writer lock. `tis_rwlock_destroy` frees up readers/writer lock internal resources.

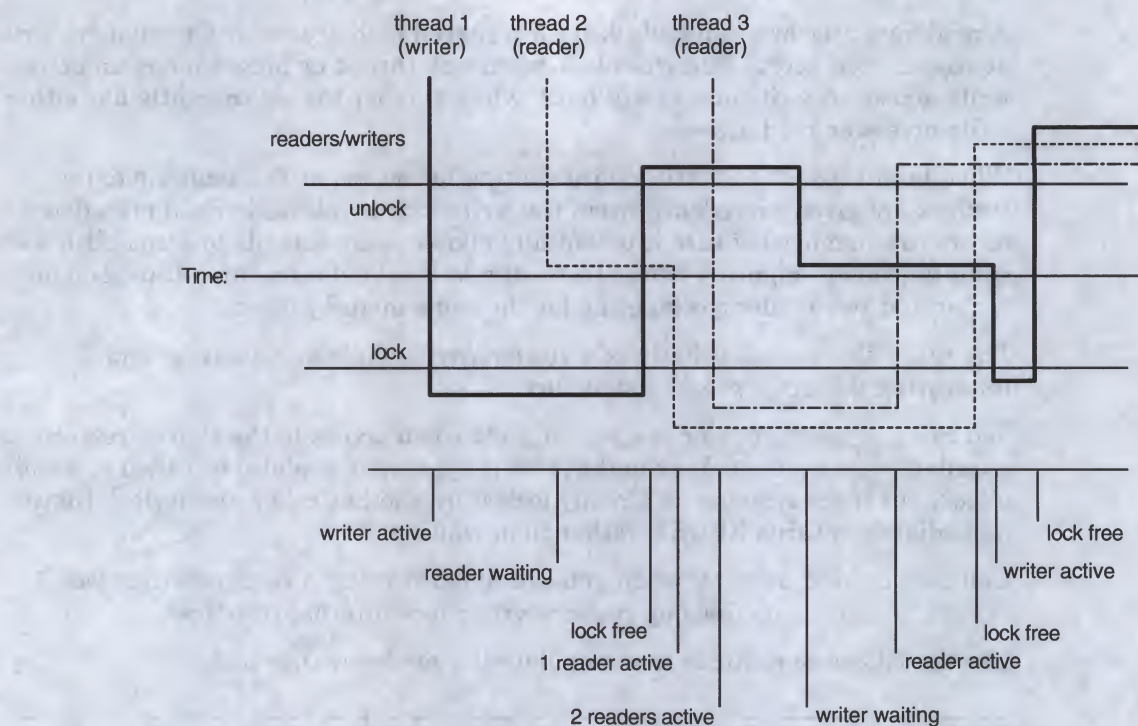
Use the following routines to manipulate the readers/writer locks:

Routine	Description
<code>tis_rwlock_init</code>	Initializes a readers/writer lock.
<code>tis_rwlock_destroy</code>	Destroys a readers/writer lock.
<code>tis_read_lock</code>	Acquires a read lock.
<code>tis_write_lock</code>	Acquires a write lock.
<code>tis_read_trylock</code>	Tries to acquire a read lock without waiting.
<code>tis_write_trylock</code>	Tries to acquire a write lock without waiting.
<code>tis_read_unlock</code>	Unlocks the read lock.
<code>tis_write_unlock</code>	Unlocks the write lock.

For more information about each readers/writer `tis` routine, see Part III.



**Figure 4-1 Readers/Writer Lock Behavior**



ZK-7929A-GE



## Using the DECthreads Exception Package

This chapter introduces conventions for the modular use of exceptions.

Although the POSIX 1003.1c interface reports errors by returning non-zero failure codes, DECthreads uses exceptions in the following cases:

- The pthread exit function raises an exception to allow modular cleanup.
- Cancellation of a thread results in raising an exception to allow modular cleanup.
- On Digital UNIX, the synchronous signals (for example, SIGSEGV) are converted to exceptions unless a signal action is declared.

The DECthreads exception package is most useful when you are programming in the C language or for other cross-platform routines using the pthread interface. On OpenVMS systems, you can use the OpenVMS condition-handling facility to catch exceptions. On Digital UNIX systems, you can use the C/C++ language exception library to catch exceptions.

### 5.1 Overview of Exceptions

An **exception** is an object that describes an error condition. Operations on exception objects allow errors to be reported and handled. If an exception is handled properly, the program can recover from errors. For example, if an exception is raised from a parity error while reading a tape, the recovery action might be to retry 100 times before giving up.

Using a few simple macros, C functions can declare a block of code (an **exception scope**) where exceptions are to be caught, and can define a block of code within an exception scope to process a specific exception (or all exceptions). DECthreads exception handlers are **attached**, which means that the handler code appears within the block where exceptions are caught. This allows you to see what actions will be taken when an exception occurs.

There are two ways to process an exception that occurs within the exception scope:

- The exception is **caught**. This means that the code handles all effects of the error and continues normal operation.
- The scope is **finalized**. This means that the current context is cleaned up and resources (such as mutexes) are released. The exception is then passed on to outer scopes for further processing. Additionally, finalization occurs even if no exception was raised so that resources are always released without duplication of code.



### 5.1.1 Types of Exceptions

There are two types of exceptions: **address** exceptions and **status** exceptions. An exception is initialized as an address exception, but it can be modified (before it is used) by defining a status value for it. Following are the primary differences between address and status exceptions:

- Different exception objects can be declared with the same status value, and those exceptions are considered identical by the exception package. For example, if one exception is raised, it can be caught by specifying another exception object having the same status. Two different address exceptions can never match each other.
- If the platform supports a universal definition of error status, then status exceptions can often be used to import and export system status values. When a facility called by DECthreads raises a system exception, DECthreads and its clients can catch the exception using a DECthreads status exception. Similarly, when a function raises a DECthreads exception, a caller might be able to handle it using facilities provided by the language or platform.

Status values used in exceptions can be interpreted, handled, and reported in a universal manner, regardless of which facility defined the status value. Use address exceptions if your code does not have a range of status codes assigned to it. Address exceptions are always unique so you do not risk colliding with another facility's status codes and inadvertently handling the wrong exception. Also, address exceptions are more portable because status codes are likely to be different on each platform.

### 5.1.2 Terminating Exception Semantics

DECthreads exceptions are terminating exceptions. This means that control never returns to the instruction following a **RAISE** statement. When an exception occurs because of a hardware condition such as an illegal address, execution cannot be resumed at the failing instruction. An exception causes execution of handlers that have been declared (starting with the most recently declared handler and proceeding backwards) until a **CATCH** or **CATCH\_ALL** clause is reached that does not end with **RERAISE**. At this point, execution continues at the first statement beyond the **ENDTRY** that terminates that current handler.

---

#### Note

---

On OpenVMS systems, all exceptions are of **SEVERE (FATAL)** severity. When you set the status of an exception using `pthread_exc_set_status_np` DECthreads will set the severity field to **SEVERE (4)**. Also, DECthreads raises exceptions (**RAISE** or `pthread_exc_raise_status_np`) using **LIB\$STOP**, that also sets the severity to **SEVERE**. Finally, the **CATCH**, **CATCH\_ALL**, and **FINALLY** macros cannot be used to handle any status exception that is not a **SEVERE** severity level.

---



## 5.2 Exception Operations

The DECthreads Exception Package allows you to perform the following operations on exceptions:

- Declare and initialize an exception object
- Raise an exception
- Define a region of code over which exceptions are caught
- Catch a particular exception or all exceptions
- Reraise the current exception
- Define epilogue actions for a block
- Import a system-defined error status into the program as an exception
- Extract a system-defined error status from an exception
- Report an exception
- Determine whether two exceptions match

These operations are discussed in the following sections.

### 5.2.1 Declaring and Initializing an Exception Object

An exception object is an opaque type that should only be manipulated by the exception package functions. The actual definition of the type may differ from one implementation to another.

Declaring and initializing an exception object documents that a program reports or handles a particular error. Having the error expressed as an exception object provides future extensibility as well as portability.

An exception is declared as a variable of type `EXCEPTION`. In general, you should declare the type as `static` or `extern`. For example:

```
static EXCEPTION an_error;
```

Because an exception object may require dynamic initialization on some platforms, the DECthreads exception package requires a run-time initialization call in addition to the declaration. The initialization function is a macro named `EXCEPTION_INIT`. The name of the exception is passed as a parameter.

Following is an example of declaring and initializing an exception object:

```
EXCEPTION parity_error;          /* Declare it */  
EXCEPTION_INIT (parity_error);  /* Initialize it */
```

### 5.2.2 Raising an Exception

Raising an exception reports an error not by returning a value, but by propagating the exception. Propagation involves searching all active scopes for code written to handle the error or code written to perform finalization actions in case of any error and causing that code to execute. If a scope does not define a handler or finalization block, then the scope is simply torn down as the exception propagates up the stack. This is sometimes referred to as **unwinding** the stack. Because DECthreads exceptions are terminating, there is no option to make execution resume at the point of the error. (Execution resumes at the point where the exception is caught.)



If an exception is unhandled, the process is terminated. Note that on OpenVMS systems, DECthreads exceptions are raised using LIB\$STOP, which always sets the condition code to a level of SEVERE (4). Termination prevents the unhandled error from affecting other areas of the program.

An example of raising an exception is as follows:

```
RAISE (parity_error);
```

### 5.2.3 Defining a Code Region to Catch Exceptions

The TRY macro defines the beginning of an exception scope, and the ENDTry macro defines the end of the scope. These macros allow the programmer to define a scope (a block) wherein exceptions can be caught. Any exceptions raised within the block, or within any functions that are called directly or indirectly by the block, pass through the control of this scope. These exceptions can be caught and reraised if it is desirable to continue propagation, or ignored which implicitly reraises them.

Following is an example of defining an exception-handling region without indicating any recovery actions:

```
TRY {  
    read_tape ();  
}  
ENDTRY
```

### 5.2.4 Catching a Particular Exception

The exception scope can express interest in any number of specific exceptions by naming them in CATCH expressions. When an exception reaches the exception scope, control is transferred to the first CATCH clause in the block that matches the exception. If there is more than one CATCH for a given exception within the scope of a single TRY/ENDTRY scope, then only the first one matching the current exception gains control.

To catch an address exception, the CATCH macro must specify the name of the exception object as used in a RAISE macro. However, status exceptions can be caught using any exception object that has been set to the same status code as the exception that was raised. In general, you should RAISE and CATCH using the same exception object even when using status exceptions.

Following is an example of catching a particular exception and specifying the recovery action (in this case, a message). After catching the exception and executing the recovery action, the exception is explicitly reraised (causing it to propagate to its callers):

```
TRY {  
    read_tape ();  
}  
CATCH (parity_error) {  
    printf ("Oops, parity error, program terminating\n");  
    printf ("Try cleaning the heads!\n");  
    RERAISE;  
}  
ENDTRY
```



### 5.2.5 Catching All Exceptions

The exception scope can express interest in all exceptions using the `CATCH_ALL` macro. No `CATCH` macros can follow the `CATCH_ALL` macro within an exception scope.

Any exception that is caught using a `CATCH_ALL` macro should be reraised. It is inappropriate to absorb exceptions that your code is not explicitly aware of. Because you cannot necessarily predict all possible exceptions that your code might encounter, you cannot assume that your code can recover in every possible situation. Therefore, your `CATCH_ALL` clause should reraise all exceptions to allow an outer scope to catch this specific exception and perform the appropriate recovery.

Following is an example of the `CATCH_ALL` macro.

```
int *local_mem;
local_mem = malloc (sizeof (int));
TRY {
    operation(local_mem);      /* May raise an exception */
    free (local_mem);
}
CATCH (an_error) {
    printf ("Oops; caught one!\n");
    free (local_mem);
    RERAISE;
}
CATCH_ALL {
    free (local_mem);
    RERAISE;
}
ENDTRY
```

### 5.2.6 Reraising the Current Exception

Within the code block of a `CATCH` or `CATCH_ALL` macro, you can use `RERAISE` to allow outer exception scopes the chance to handle the exception. Do this when the current scope needs to restore some permanent state (for example, releasing resources such as memory or a mutex), but does not have enough context about the error to attempt to recover.

The `RERAISE` function is only valid in the code of a `CATCH` or `CATCH_ALL` clause. For example:

```
int *local_mem;
local_mem = malloc (sizeof (int));
TRY {
    operation(local_mem);      /* May raise an exception */
    free (local_mem);
}
CATCH (an_error) {
    free (local_mem);
    RERAISE;
}
```



## 5.2.7 Defining Epilogue Actions for a Block

Frequently, the only reason a block of code needs to catch exceptions is to perform cleanup actions, such as releasing resources. In many cases, the same operations are performed whether the block exits normally or with an exception; under many exception models, this requires duplicating code (both within a `CATCH_ALL` type construct, and following the exception scope in case an exception does not occur).

The `FINALLY` macro catches an exception and then reraises the exception for outer scopes to handle. The actions defined by a `FINALLY` clause are also performed when the scope exits normally without an exception, so that they do not need to be duplicated.

Do not combine the `FINALLY` clause with `CATCH` or `CATCH_ALL`. Doing so results in unpredictable behavior.

Following is an example of the `FINALLY` macro:

```
int *local_mem;
local_mem = malloc (sizeof (int));
TRY {
    operation(local_mem);      /* May raise an exception */
}
FINALLY {
    free (local_mem);
}
ENDTRY
```

## 5.2.8 Determining the Current Exception

The current exception object can be referenced within a `CATCH` or `CATCH_ALL` block by using the name `THIS_CATCH`. The exception object `THIS_CATCH` has a type of `EXCEPTION *`. This value can be passed to `pthread_exc_get_status_np()`, `pthread_exc_report_np()`, or `pthread_exc_matches_np()` (defined in Section 5.2.10, Section 5.2.11, and Section 5.2.12).

Because of the way exceptions are propagated, the address contained in `THIS_CATCH` might not be the actual address of an address exception; if you need to match `THIS_CATCH` against known exceptions, use `pthread_exc_matches_np()`.

## 5.2.9 Importing a System-Defined Error Status into the Program as an Exception

The `pthread_exc_set_status_np()` function can be used to create a status exception. The exception object must already have been initialized with `EXCEPTION_INIT`. Any system-specific status value may be used. All exception objects set to the same status value are considered equivalent by the exception facility.

An example of importing an error status into an exception is as follows:

```
void pthread_exc_set_status_np (EXCEPTION *exception,
                               unsigned int code);

static EXCEPTION an_error;

EXCEPTION_INIT (an_error);
pthread_exc_set_status_np (&an_error, ENOMEM);
```

---

### Note

On OpenVMS systems, DECthreads exception status values must



always have a SEVERE severity level. The `pthread_exc_set_status_np` operation will modify (if necessary) the severity level of the status code.

---

### 5.2.10 Exporting a System-Defined Error Status

The `pthread_exc_get_status_np()` function can be used to retrieve the system status value from a status exception, for example, after an exception is caught. If the exception object specified is a status exception, `pthread_exc_get_status_np()` sets the status value argument and returns 0; otherwise, it returns `EINVAL` and does not set the status value argument. For example:

```
int pthread_exc_get_status_np (EXCEPTION *exception,
                              unsigned int *status);

TRY {
    operation ();
}
CATCH_ALL {
    int status;
    if (pthread_exc_get_status_np (THIS_CATCH, &status) == 0
        && status < sys_nerr)
        fprintf (stderr, "%Exception %s\n",
                 strerror (status));
    else
        pthread_exc_report_np (THIS_CATCH);
}
ENDTRY
```

### 5.2.11 Reporting an Exception

DECthreads reports an exception only when it is raised without a `CATCH` or `CATCH_ALL`, immediately before the process is terminated. Sometimes client code might wish to report an exception as part of error recovery. The `pthread_exc_report_np()` function prints a message to `stderr` or `SYSSERROR` describing the exception.

All predefined exceptions have an associated message describing the error. Normally, when the DECthreads exception package has been well-integrated with a platform status mechanism, external status values can also be reported. However, when an address exception is reported, DECthreads can only report the fact that an exception has occurred, and the address of the exception object.

Following is an example of reporting an error:

```
void pthread_exc_report_np (EXCEPTION *exception);
```

For example:

```
pthread_exc_report_np (&illinstr);
```

### 5.2.12 Determining Whether Two Exceptions Match

The `pthread_exc_matches_np()` function compares two exception objects, taking into consideration whether they are address or status exceptions, and possibly other system-specific rules for matching status values. Whenever you need to compare two exceptions, you should use this function. For example:



```

int pthread_exc_matches_np (EXCEPTION *exception1,
                           EXCEPTION *exception2);
EXCEPTION my_status;
EXCEPTION_INIT (&my_status);
pthread_exc_set_status_np (&my_status, status_code);
.
.
.
if (pthread_exc_matches_np (THIS_CATCH, &my_status))
    fprintf (stderr, "This is my exception\n");

```

## 5.3 C Language Syntax

The following example shows the syntax for handling exceptions:

```

TRY
    try_block
    [CATCH (exception_name)
        handler_block]...
    [CATCH_ALL
        handler_block]
ENDTRY

```

A **try\_block** or a **handler\_block** is a sequence of statements, the first of which may be declarations, as in a normal block. If an exception is raised in the **try\_block**, the catch clauses are evaluated to see if any one matches the current exception.

The **CATCH** or **CATCH\_ALL** clauses absorb an exception—they can catch an exception propagating out of the **try\_block** and direct execution into the associated **handler\_block**. Propagation of the exception, by default, then ends. Within the lexical scope of a handler, it is possible to cause propagation of the same exception to resume (this is called **raising** the exception), or it is possible to raise some new exception.

The **RERAISE** statement is allowed in any handler statements and causes the current exception to be **reraised**. Propagation of the caught exception resumes.

The **RAISE (exception\_name)** statement is allowed anywhere and causes a particular exception to start propagating. For example:

```

TRY
    sort(); /* Call a function that may raise an exception.
            * An exception propagates by transferring control
            * out of some nested routine back to the TRY
            * clause. Any output parameters or return values
            * of the called routine are therefore indeterminate.
            */

    CATCH (pthread_cancel_e)
        printf("Canceled while sorting\n");
        RERAISE;

    CATCH_ALL
        printf("Some other exception while sorting\n");
        RERAISE;

ENDTRY

```



In the previous example, if the `pthread_cancel_e` exception propagates out of the function call, the first `printf` is executed. If any other exception propagates out of sort, the second `printf` is executed. In either situation, propagation of the exception resumes because of the `RERAISE` statement. (If the code is unable to fully recover from the error, or does not understand the error, it needs to further propagate the error to its callers, as in the previous example.)

The following example shows the syntax for an epilogue:

```
TRY
    try_block
FINALLY
    final_block
ENDTRY
```

The `final_block` is executed regardless of whether the `try_block` executes to completion without raising an exception or if an exception is raised in the `try_block`. If an exception is raised in the `try_block`, propagation of the exception is resumed after executing the `final_block`.

Note that a `CATCH_ALL` handler and `RERAISE` could be used to do this, but the epilogue code would then have to be duplicated in two places, as follows:

```
TRY
    try_block
CATCH_ALL
    final_block
    RERAISE;
ENDTRY
{ final_block }
```

A `FINALLY` statement has exactly this meaning but avoids code duplication.

---

#### Note

---

The behavior of `FINALLY` along with `CATCH` or `CATCH_ALL` clauses is unpredictable. Do not combine them for the same `try_block`.

---

Another example of the `FINALLY` statement is as follows:

```
pthread_mutex_lock (some_object.mutex);
some_object.num_waiters = some_object.num_waiters + 1;
TRY
    while (! some_object.data_available)
        pthread_cond_wait (some_object.condition);
    /* The code to act on the data_available goes here */
FINALLY
    some_object.num_waiters = some_object.num_waiters - 1;
    pthread_mutex_unlock (some_object.mutex);
ENDTRY
```

In the previous example, the call to `pthread_cond_wait` could raise the `pthread_cancel_e` exception if the thread was canceled while it was waiting. The `final_block` ensures that the shared data associated with the lock is correct for the next thread that acquires the mutex.



## 5.4 Rules and Conventions for Modular Use of Exceptions

The following rules ensure that exceptions are used in a modular way (so that independent software components can be written without requiring knowledge of each other):

- Use unique names for exceptions.

A naming convention ensures that the names for exceptions that are declared `EXTERN` from different modules do not clash. The following convention is recommended.

`<facility-prefix>_<error-name>_e`

Example: `pthread_cancel_e`

- Avoid putting code in a `TRY` macro that belongs before it.

The `TRY` macro should only guard statements for which the statements in the `FINALLY` or `CATCH`, or `CATCH_ALL` clauses are always valid to execute.

A common misuse of `TRY` is to put code in the `try_block` that should be placed before `TRY`. An example of this misuse is as follows:

```
TRY
    handle = open_file (file_name);
    /* Statements that may raise an exception here */
FINALLY
    close (handle);
ENDTRY
```

The previous `FINALLY` code assumes that no exception is raised by `open_file`. Otherwise, the code would access an invalid identifier in the `FINALLY` part if `open_file` is modified to raise an exception. The previous example should be rewritten as follows:

```
handle = open_file (file_name);
TRY
    /* Statements that may raise an exception here */
FINALLY
    close (handle);
ENDTRY
```

The code that is an opening bracket belongs prior to `TRY`, and the code that is its matching closing bracket belongs in the `FINALLY` clause.

- Raise exceptions prior to performing side-effects.

Write functions that propagate exceptions to their callers so that the function does not modify any persistent process state before raising the exception. A call to the matching close call is required only if the open operation is successful. (If an exception is raised, the caller cannot access the output parameters of the function because the compiler may not have copied temporary values back to their home locations from registers.)

If `open_file` raises an exception, the identifier will not have been written, so `open` must not require that `close` be called when `open` raises an exception. This property is also what allows the call to be moved to `open_file` prior to the `TRY`.

- Do not place a return or go-to between `TRY` and `ENDTRY`.

It is invalid to return or go-to or leave by some other means a `TRY`, `CATCH`, `CATCH_ALL`, or `FINALLY` block. Special code is generated by the `ENDTRY` macro and it must be executed.



- Use the ANSI C volatile attribute.  
Variables that are read or written by exception handling code must be declared with the ANSI C volatile attribute. Run your tests with the optimize compiler option to ensure that the compiler thoroughly tests your exception handling code.
- Reraise exceptions that are not fully handled.  
Reraise any exception that you catch, unless your handler has performed the complete recovery action for the error. This rule permits an unhandled exception to propagate to some final default handler that knows how to recover fully.  
A corollary of this rule is that CATCH\_ALL handlers must reraise, since they may catch any exception, and usually cannot do recovery actions that are proper for every exception.  
Following this convention is important so that you also do not absorb a cancel or thread-exit request. These are mapped into exceptions so that exception handling has the full power to handle all exceptional conditions, from access violations to thread-exit. (In some applications it is important to be able to catch these to preserve an external invariant, such as an on-disk database.)
- Declare only static exceptions.

## 5.5 DECthreads Exceptions and Definitions

Table 5-1 lists the DECthreads exceptions and briefly explains the meaning of each exception.

Exception names beginning with the prefix `pthread_` or `cma_` are raised as the result of something happening internal to the DECthreads facility and are not meant to be raised by user code. Exceptions beginning with `pthread_exc_` are generic and belong to the exception facility and/or the underlying system.

See Section E.4 for a list and descriptions of the `cma` interface exceptions.

**Table 5-1 pthread Exceptions**

Exception	Definition
<code>pthread_exc_aritherr_e</code>	Unhandled floating-point exception signal ("arithmetic error")
<code>pthread_cancel_e</code>	Thread cancellation in progress
<code>pthread_exc_excpcu_e</code>	"cpu-time limit exceeded"
<code>pthread_exit_e</code>	Thread exited using <code>pthread_exit_e()</code>
<code>pthread_exc_exflsiz_e</code>	"File size limit exceeded"
<code>pthread_stackovf_e</code>	Attempted stack overflow was detected
<code>pthread_exc_decovf_e</code>	Unhandled decimal overflow trap exception
<code>pthread_exc_exquota_e</code>	Operation failed due to insufficient quota
<code>pthread_exc_fltdiv_e</code>	Unhandled floating-point/decimal divide by zero trap exception
<code>pthread_exc_fltovf_e</code>	Unhandled floating-point overflow trap exception

(continued on next page)



**Table 5-1 (Cont.) pthread Exceptions**

Exception	Definition
pthread_exc_fltund_e	Unhandled floating-point underflow trap exception
pthread_exc_illaddr_e	Data or object could not be referenced
pthread_exc_illinstr_e	Unhandled illegal instruction signal ("illegal instruction")
pthread_exc_insfmem_e	Insufficient virtual memory for requested operation
pthread_exc_intdiv_e	Unhandled integer divide by zero trap exception
pthread_exc_intovf_e	Unhandled integer overflow trap exception
pthread_exc_nopriv_e	Insufficient privilege for requested operation
pthread_exc_privinst_e	Unhandled privileged instruction fault exception
pthread_exc_resaddr_e	Unhandled reserved addressing fault exception
pthread_exc_resoper_e	Unhandled reserved operand fault exception
pthread_exc_SIGABRT_e	Unhandled signal ABRT
pthread_exc_SIGBUS_e	Unhandled bus error signal
pthread_exc_SIGEMT_e	Unhandled EMT signal
pthread_exc_SIGFPE_e	Unhandled floating-point exception signal
pthread_exc_SIGILL_e	Unhandled illegal instruction signal
pthread_exc_SIGIOT_e	Unhandled IOT signal
pthread_exc_SIGPIPE_e	Unhandled broken pipe signal
pthread_exc_SIGSEGV_e	Unhandled segmentation violation signal
pthread_exc_SIGSYS_e	Unhandled bad system call signal
pthread_exc_SIGTRAP_e	Unhandled trace or breakpoint trap signal
pthread_exc_subrng_e	Unhandled subscript out of range exception
pthread_exc_uninitexc_e	Uninitialized exception raised



---

## DECthreads Example

This chapter presents an example that shows the use of the DECthreads pthread routines from a program written in the C language. Example 6-1 uses the status-returning interface (the default) to perform a prime number search.

### 6.1 Prime Number Search Example

Example 6-1 shows the use of the DECthreads pthread routines in a C program that performs a prime number search. The program finds a specified number of prime numbers, then sorts and displays these numbers. Several threads participate in the search: each thread takes a number (the next one to be checked), checks if it is a prime, records it if it is prime, and then takes another number—and so on.

This program shows the work crew model of programming (see Section 1.4.2.) The workers (threads) increment a number (`current_num`) to get their next work assignment. As a whole, the worker threads are responsible for finding a specified number of prime numbers, at which point their work is completed.

The number of workers to be used and the requested number of prime numbers to be found are defined constants. A macro is used to check for error status and to print a given string and the associated error value. Data to be accessed by all threads (mutexes, condition variables, and so forth) are declared as global items.

Worker threads execute the prime search routine, which begins by synchronizing with the parent thread utilizing a predicate and a condition variable. Enclose a condition wait in a predicate loop to prevent a thread from continuing if it is wrongly signaled or broadcasted. The lock associated with the condition variable must be held by the thread during the call to condition wait. The lock is released within the call and acquired again upon being signaled or broadcasted. The same mutex must be used for all operations performed on a specific condition variable.

After the parent sets the predicate and broadcasts, the workers begin finding prime numbers until canceled by a fellow worker who has found the last requested prime number. Upon each iteration the workers increment the current number to be worked on and take the new value as their work item. A mutex is locked and unlocked around getting the next work item, in order to ensure that no two threads are working on the same item. This type of locking protocol should be performed on all global data to ensure its integrity.

Each worker thread then determines if its current work item (number) is prime by trying to divide numbers into it. If the number proves to be nondivisible, it is put on the list of primes. Cancels are turned off while working with the list of primes to better control any cancels that do occur. The list of primes and its current count are protected by locks, which also protect the cancellation process of all other worker threads upon finding the last requested prime. While still under the prime list lock, the current worker checks to see if it has found the last requested prime, and, if so, unsets a predicate and cancels all other worker



threads. Cancels are then reenabled. The canceling thread should fall out of the work loop as a result of the predicate that it unsets.

The parent thread's flow of execution is as follows:

- Set up the environment.

Setting up the environment means initializing mutexes and the one condition variable used in the example.

- Create worker threads.

Creation of worker threads is straightforward and utilizes the default attributes. Worker threads immediately wait on a condition variable.

- Broadcast to the worker threads that they may start.

- Join each thread as it finishes.

As the parent joins each of the returning worker threads, it receives an exit value from them that indicates whether a thread exited normally or not. In this case the exit values on all but one of the worker threads should be -1, indicating that they were canceled.

- Sort and print the list of primes.

The following pthread routines are used in Example 6-1:

```
pthread_cancel
pthread_cond_broadcast
pthread_cond_init
pthread_cond_wait
pthread_create
pthread_detach
pthread_exit
pthread_join
pthread_mutex_init
pthread_mutex_lock
pthread_mutex_unlock
pthread_setcancelstate
pthread_setcanceltype
pthread_testcancel
```

#### Example 6-1 C Program Example (Prime Number Search)

```
/*
 *
 * DECthreads example program conducting a prime number search
 *
 */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

/*
 * Constants used by the example.
 */
#define workers 5 /* Threads to perform prime check */
#define request 110 /* Number of primes to find */
```

(continued on next page)



### Example 6-1 (Cont.) C Program Example (Prime Number Search)

```
/*
 * Macros
 */

#define check(status,string) \
    if (status != 0) perror (string)

/*
 * Global data
 */

pthread_mutex_t prime_list; /* Mutex for use in accessing the prime */
pthread_mutex_t current_mutex; /* Mutex associated with current number */
pthread_mutex_t cond_mutex; /* Mutex used for ensuring CV integrity */
pthread_cond_t cond_var; /* Condition variable for thread start */
int current_num= -1; /* Next number to be checked, start odd */
int thread_hold=1; /* Number associated with condition state */
int count=0; /* Count of prime numbers - index to primes */
int primes[request]; /* Store prime numbers - synchronize access */
pthread_t threads[workers]; /* Array of worker threads */
int orig_c_s; /* Original cancel state */
int work_c_s; /* Working cancel state */
int s; /* Cancel routine status */

static void
unlock_cond (void *arg)
{
    int status; /* Hold status from pthread calls */

    status = pthread_mutex_unlock (&cond_mutex);
    check(status,"3:Mutex_unlock bad status\n");
}

/*
 * Worker thread routine.
 *
 * Worker threads start with this routine, which begins with a condition
 * wait designed to synchronize the workers and the parent. Each worker
 * thread then takes a turn taking a number for which it will determine
 * whether or not it is prime.
 */

static void *
prime_search (void *arg)
{
    div_t div_results; /* DIV results: quot and rem */
    int numerator; /* Used for determining primeness */
    int denominator; /* Used for determining primeness */
    int cut_off; /* Number being checked div 2 */
    int notifiee; /* Used during a cancellation */
    int prime; /* Flag used to indicate primeness */
    int my_number; /* Worker thread identifier */
    int status; /* Hold status from pthread calls */
    int not_done=1; /* Work loop predicate */
    my_number = (int)arg;

    /*
     * Synchronize threads and the parent using a condition variable, of
     * which the predicate (thread_hold) will be set by the parent.
     */
}
```

(continued on next page)



### Example 6-1 (Cont.) C Program Example (Prime Number Search)

```
status = pthread_mutex_lock (&cond_mutex);
check(status,"1:Mutex_lock bad status\n");

pthread_cleanup_push (unlock_cond, NULL);

    while (thread_hold) {
        status = pthread_cond_wait (&cond_var, &cond_mutex);
        check(status,"2:Cond_wait bad status\n");
    }

pthread_cleanup_pop (1);

/*
 * Perform checks on ever larger integers until the requested
 * number of primes is found.
 */

while (not_done) {

    /* Cancellation point */
    pthread_testcancel ();

    /* Get next integer to be checked */
    status = pthread_mutex_lock (&current_mutex);
    check(status,"4:Mutex_lock bad status\n");
    current_num = current_num + 2;          /* Skip even numbers */
    numerator = current_num;
    status = pthread_mutex_unlock (&current_mutex);
    check(status,"5:Mutex_unlock bad status\n");

    /* Only need to divide in half if number to verify not prime */
    cut_off = numerator/2 + 1;
    prime = 1;

    /* Check for prime; exit if something evenly divides */
    for (denominator = 2; ((denominator < cut_off) && (prime));
        denominator++) {
        prime = numerator % denominator;
    }

    if (prime != 0) {

        /* Explicitly turn off all cancels */
        s = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &orig_c_s);

        /*
         * Lock a mutex and add this prime number to the list. Also,
         * if this fulfills the request, cancel all other threads.
         */

        status = pthread_mutex_lock (&prime_list);
        check(status,"6:Mutex_lock bad status\n");
```

(continued on next page)



### Example 6-1 (Cont.) C Program Example (Prime Number Search)

```
        if (count < request) {
            primes[count] = numerator;
            count++;
        }
        else if (count == request) {
            not_done = 0;
            count++;
            for (notifiee = 0; notifiee < workers; notifiee++) {
                if (notifiee != my_number) {
                    status = pthread_cancel ( threads[notifiee] );
                    check(status,"12:Cancel bad status\n");
                }
            }

            status = pthread_mutex_unlock (&prime_list);
            check(status,"13:Mutex_unlock bad status\n");

            /* Reenable cancels */
            s = pthread_setcancelstate(orig_c_s, &work_c_s);
        }

        pthread_testcancel ();
    }

    pthread_exit ((void *)my_number);
    return (void *)0;
}

main()
{
    int    worker_num;      /* Counter used when indexing workers */
    void   *exit_value;     /* Individual worker's return status */
    int    list;            /* Used to print list of found primes */
    int    status;          /* Hold status from pthread calls */
    int    index1;          /* Used in sorting prime numbers */
    int    index2;          /* Used in sorting prime numbers */
    int    temp;            /* Used in a swap; part of sort */
    int    not_done;        /* Indicates swap made in sort */

    /*
     * Create mutexes
     */

    status = pthread_mutex_init (&prime_list, NULL);
    check(status,"7:Mutex_init bad status\n");
    status = pthread_mutex_init (&cond_mutex, NULL);
    check(status,"8:Mutex_init bad status\n");
    status = pthread_mutex_init (&current_mutex, NULL);
    check(status,"9:Mutex_init bad status\n");

    /*
     * Create condition variable
     */

    status = pthread_cond_init (&cond_var, NULL);
    check(status,"10:Cond_init bad status\n");

    /*
     * Create the worker threads.
     */
}
```

(continued on next page)



### Example 6-1 (Cont.) C Program Example (Prime Number Search)

```
for (worker_num = 0; worker_num < workers; worker_num++) {
    status = pthread_create (
        &threads[worker_num],
        NULL,
        prime_search,
        (void *)worker_num);
    check(status, "11: Pthread_create bad status\n");
}

/*
 * Set the predicate thread_hold to zero, and broadcast on the
 * condition variable that the worker threads may proceed.
 */

status = pthread_mutex_lock (&cond_mutex);
check(status, "12: Mutex_lock bad status\n");

thread_hold = 0;
status = pthread_cond_broadcast (&cond_var);

status = pthread_mutex_unlock (&cond_mutex);
check(status, "13: Mutex_unlock bad status\n");

/*
 * Join each of the worker threads in order to obtain their
 * summation totals, and to ensure each has completed
 * successfully.
 */

/*
 * Mark thread storage free to be reclaimed upon termination by
 * detaching it.
 */

for (worker_num = 0; worker_num < workers; worker_num++) {
    status = pthread_join (
        threads[worker_num],
        &exit_value );
    check(status, "14: Pthread_join bad status\n");

    if (exit_value == (void *)worker_num)
        printf("Thread terminated normally\n");
}

/*
 * Upon normal termination the exit_value is equivalent to
 * worker_num.
 */

status = pthread_detach ( threads[worker_num] );
check(status, "15: Pthread_detach bad status\n");
}

/*
 * Take the list of prime numbers found by the worker threads and
 * sort them from lowest value to highest. The worker threads work
 * concurrently; there is no guarantee that the prime numbers
 * will be found in order. Therefore, a sort is performed.
 */
```

(continued on next page)



### Example 6-1 (Cont.) C Program Example (Prime Number Search)

```
not_done = 1;
for (index1 = 1; ((index1 < request) && (not_done)); index1++) {
    for (index2 = 0; index2 < index1; index2++) {
        if (primes[index1] < primes[index2]) {
            temp = primes[index2];
            primes[index2] = primes[index1];
            primes[index1] = temp;
            not_done = 0;
        }
    }
}

/*
 * Print out the list of prime numbers that the worker threads
 * found.
 */
printf ("The list of %d primes follows:\n", request);
printf ("%d", primes[0]);
for (list = 1; list < request; list++) {
    printf (" ,\t%d", primes[list]);
}
printf ("\n");
}
```



# Example 1 (Cont.) - Finding Extrema (in a smaller domain)

Let  $f(x, y) = x^2 + y^2 - 2x - 4y + 3$ . Find the absolute maximum and minimum values of  $f$  on the domain  $D = \{(x, y) \mid x^2 + y^2 \leq 4\}$ .

**Solution:** The domain  $D$  is a disk of radius 2 centered at the origin.

First, find the critical points of  $f$  in the interior of  $D$ . The gradient of  $f$  is  $\nabla f(x, y) = (2x - 2, 2y - 4)$ . Setting  $\nabla f(x, y) = (0, 0)$  gives the system of equations  $2x - 2 = 0$  and  $2y - 4 = 0$ , which has the unique solution  $(1, 2)$ . Since  $(1, 2)$  is not in  $D$ , there are no critical points of  $f$  in the interior of  $D$ .



# Part II

---

## POSIX 1003.1c (pthread) Routines Reference

Part II provides detailed descriptions of routines that make up the DECthreads POSIX interface. These routines, prefixed **pthread**, are the DECthreads implementation of the POSIX Standard 1003.1c-1995 that is sponsored by the IEEE Computer Society.

---

### Note

---

The pthread routines described in this guide are based on the final POSIX 1003.1c-1995 standard approved by the IEEE. DECthreads users should be aware that applications written to the 1003.4a Draft 4 interfaces will require significant modifications to upgrade to the new interfaces.

---

Note that *errno* is NOT used by these routines. To indicate errors, the pthread routines return integer values indicating the type of error.

Routine names ending with the *\_np* suffix denote that the routine is not portable - the routine might not be available in implementations of POSIX 1003.1c other than DECthreads.



# Part II

## POST-1983 to (present) Financial Performance

The following table shows the financial performance of the company from 1983 to the present. The table is divided into two main sections: 'Pre-1983' and 'Post-1983'. The 'Pre-1983' section shows the company's performance from 1983 to 1989, while the 'Post-1983' section shows the company's performance from 1990 to the present. The table includes data on sales, profit, and other financial indicators.

The company's financial performance has improved significantly since 1983. Sales have increased from £10 million in 1983 to £50 million in 1990, and profit has increased from £1 million in 1983 to £5 million in 1990. This improvement is due to a number of factors, including increased sales, improved cost control, and a more efficient production process.

The company's financial performance has continued to improve since 1990. Sales have increased from £50 million in 1990 to £100 million in 1995, and profit has increased from £5 million in 1990 to £10 million in 1995. This improvement is due to a number of factors, including increased sales, improved cost control, and a more efficient production process.



## pthread\_atfork

Declares handlers to be called when the process forks a child.

*This routine is for Digital UNIX systems only.*

### Syntax

```
pthread_atfork(
    prepare,
    parent,
    child );
```

Argument	Data Type	Access
prepare	Handler	read
parent	Handler	read
child	Handler	read

### C Binding

```
#include <pthread.h>

int
pthread_atfork (
    void    (*prepare)(void),
    void    (*parent)(void),
    void    (*child)(void) );
```

### Arguments

#### prepare

Address of a procedure that performs the fork preparation handling. This routine is called in the parent process before creating the child.

#### parent

Address of a procedure that performs the fork parent handling. This routine is called in the parent process after creating the child and before returning to the caller of fork(2).

#### child

Address of a procedure that performs the fork child handling. This routine is called in the child process before returning to the caller of fork(2).

### Description

This routine allows a main program or library to control resources during a UNIX fork(2) operation by declaring fork handlers that are called before and after the fork(2) execution. The *prepare* fork handler is called before the fork(2) execution. The parent handler is called after the fork(2) execution within the parent process, and the child handler is called after fork(2) in the new child process. If no handling is desired, you can set any of the arguments to a NULL.



---

### Note

---

It is not legal to call `pthread_atfork` from within a handler routine. Doing so could cause a deadlock.

---

Mutex locks might be held by threads that no longer exist in a child process and any associated state might be inconsistent. The program can avoid this problem by calling `pthread_atfork()` to provide code that acquires and releases locks critical to the child process. For example, if your library uses a mutex named "my\_mutex", you might provide `pthread_atfork` handlers like:

```
void my_prepare(void)
{
    pthread_mutex_lock(&my_mutex);
}

void my_parent(void)
{
    pthread_mutex_unlock(&my_mutex);
}

void my_child(void)
{
    pthread_mutex_unlock(&my_mutex);
    /* Reinitialize state that doesn't apply...like heap owned */
    /* by other threads */
}

{
    .
    .
    .
    pthread_atfork(my_prepare, my_parent, my_child);
    .
    .
    fork();
}
```

### Return Values

If an error occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[ENOMEM]	Insufficient table space exists to record the fork handler addresses.



## pthread\_attr\_destroy

Destroys a thread attributes object.

### Syntax

```
pthread_attr_destroy(
    attr );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	modify

### C Binding

```
#include <pthread.h>

int
pthread_attr_destroy (
    pthread_attr_t  *attr);
```

### Arguments

**attr**  
Thread attributes object to be destroyed.

### Description

This routine destroys a thread attributes object. Call this routine when a thread attributes object will no longer be referenced.

Threads that were created using this thread attributes object are not affected by the destruction of the thread attributes object.

The results of calling this routine are unpredictable if the value specified by the *attr* argument refers to a thread attributes object that does not exist.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>attr</i> is invalid.

### Associated Routines

pthread\_attr\_init



## pthread\_attr\_getdetachstate

## pthread\_attr\_getdetachstate

Obtains the detachstate attribute of the specified thread attributes object.

### Syntax

```
pthread_attr_getdetachstate(  
    attr,  
    detachstate );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
detachstate	integer	write

### C Binding

```
#include <pthread.h>  
  
int  
pthread_attr_getdetachstate (  
    const pthread_attr_t  *attr,  
    int  *detachstate);
```

### Arguments

#### attr

Thread attributes object whose detachstate attribute is obtained.

#### detachstate

Receives the value of the *detachstate* attribute.

### Description

This routine obtains the detachstate attribute of thread creation. This attribute specifies whether threads created using the specified thread attributes object are created in detached state.

On successful completion, this routine returns a zero and the detached state attribute is set in *detachstate*. A value of `PTHREAD_CREATE_JOINABLE` indicates the thread is not detached, and a value of `PTHREAD_CREATE_DETACHED` indicates the thread is detached.

See the `pthread_attr_setdetachstate` description for information about the detachstate attribute.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.



Return	Description
[EINVAL]	The value specified by <i>attr</i> does not refer to an existing thread attributes object.

**See Also**

pthread\_attr\_init  
pthread\_attr\_setdetachstate



## pthread\_attr\_getguardsize\_np

### pthread\_attr\_getguardsize\_np

Obtains the guardsize attribute of the specified thread attributes object.

#### Syntax

```
pthread_attr_getguardsize_np(  
    attr,  
    guardsize );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
guardsize	size_t	write

#### C Binding

```
#include <pthread.h>  
  
int  
pthread_attr_getguardsize_np (  
    const pthread_attr_t  *attr  
    size_t  *guardsize);
```

#### Arguments

##### attr

Thread attributes object whose guardsize attribute is obtained.

##### guardsize

Receives the value for the guardsize attribute. The *guardsize* argument specifies the minimum size (in bytes) of the guard area for a thread.

#### Description

This routine obtains the minimum size (in bytes) of the guard area for the stack of a thread that is created using the attributes object specified by the *attr* argument.

A guard area helps to detect stack overflows by preventing memory access beyond the thread's stack. Large guard areas are necessary when threads might allocate large structures on the stack.

#### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>attr</i> is invalid.



## Associated Routines

pthread\_attr\_init  
 pthread\_attr\_setstacksize  
 pthread\_attr\_setguardsize\_np  
 pthread\_create



---

## pthread\_attr\_getinheritsched

Obtains the inherit scheduling attribute of the specified thread attributes object.

### Syntax

```
pthread_attr_getinheritsched(  
    attr,  
    inheritsched );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
inheritsched	integer	write

### C Binding

```
#include <pthread.h>  
  
int  
pthread_attr_getinheritsched (  
    const pthread_attr_t  *attr  
    int  *inheritsched);
```

### Arguments

#### attr

Thread attributes object whose inherit scheduling attribute is obtained.

#### inheritsched

Receives the value of the inherit scheduling attribute. Refer to the description of the pthread\_attr\_setinheritsched function for valid values.

### Description

This routine obtains the value of the inherit scheduling attribute from the specified thread attributes object. The inherit scheduling attribute specifies whether threads created using the attributes object inherit the scheduling attributes of the creating thread, or use the scheduling attributes stored in the attributes object that is passed to pthread\_create.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>attr</i> is invalid.



## Associated Routines

pthread\_attr\_init  
 pthread\_attr\_setinheritsched  
 pthread\_create



---

## pthread\_attr\_getschedparam

Obtains the scheduling parameters for an attribute of the specified thread attributes object.

### Syntax

```
pthread_attr_getschedparam(  
    attr,  
    param );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
param	struct sched_param	write

### C Binding

```
#include <pthread.h>  
  
int  
pthread_attr_getschedparam (  
    const pthread_attr_t *attr,  
    struct sched_param *param);
```

### Arguments

**attr**

Thread attributes object of the scheduling policy attribute whose parameters are obtained.

**param**

Receives the values of scheduling parameters for the scheduling policy attribute of the attributes object specified by the *attr* argument. Refer to the description of the pthread\_attr\_setschedparam function for valid parameters and their values.

### Description

This routine obtains the scheduling parameters associated with the scheduling policy attribute of the specified thread attributes object.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>attr</i> is invalid.



## Associated Routines

pthread\_attr\_init  
 pthread\_attr\_setschedparam  
 pthread\_create



## pthread\_attr\_getschedpolicy

---

## pthread\_attr\_getschedpolicy

Obtains the scheduling policy attribute of the specified thread attributes object.

### Syntax

```
pthread_attr_getschedpolicy(  
    attr,  
    policy );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
policy	integer	write

### C Binding

```
#include <pthread.h>  
  
int  
pthread_attr_getschedpolicy (  
    const pthread_attr_t  *attr,  
    int  *policy);
```

### Arguments

#### attr

Thread attributes object whose scheduling policy attribute is obtained.

#### policy

Receives the value of the scheduling policy attribute. Refer to the description of the pthread\_attr\_setschedpolicy function for valid values.

### Description

This routine obtains the value of the scheduling policy attribute of the specified thread attributes object. The scheduling policy attribute defines the scheduling policy for threads created using the attributes object.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>attr</i> is invalid.



**Associated Routines**

pthread\_attr\_init  
 pthread\_attr\_setschedpolicy  
 pthread\_create



## pthread\_attr\_getstacksize

Obtains the *stacksize* attribute of the specified thread attributes object.

### Syntax

```
pthread_attr_getstacksize(
    attr,
    stacksize );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
stacksize	size_t	write

### C Binding

```
#include <pthread.h>

int
pthread_attr_getstacksize (
    const pthread_attr_t *attr,
    size_t *stacksize);
```

### Arguments

#### attr

Thread attributes object whose *stacksize* attribute is obtained.

#### stacksize

Receives the value for the *stacksize* attribute.

### Description

This routine obtains the *stacksize* attribute of a specified thread attributes (*attr*) object.

### Return Values

On successful completion, this routine returns a zero (0) and the *stacksize* value in the *stacksize* variable.

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
EINVAL	The value specified by <i>attr</i> is invalid.



## Associated Routines

pthread\_attr\_init  
 pthread\_attr\_setstacksize  
 pthread\_create



## pthread\_attr\_init

Initializes a thread attributes object.

### Syntax

```
pthread_attr_init(
    attr );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write

### C Binding

```
#include <pthread.h>

int
pthread_attr_init (
    pthread_attr_t  *attr);
```

### Arguments

**attr**  
Thread attributes object to be initialized.

### Description

This routine initializes a thread attributes object that is used to specify the attributes of threads when they are created. The attributes object created by this routine is used only in calls to `pthread_create`.

The following routines can be used to change the initialized individual attributes:

```
pthread_attr_setdetachstate
pthread_attr_setguardsize_np
pthread_attr_setinheritsched
pthread_attr_setschedparam
pthread_attr_setschedpolicy
pthread_attr_setstacksize
```

The individual attributes (internal fields) of the attributes object are initialized to default values. The default values of each attribute are discussed in the descriptions of the thread attribute setting routines listed above.

When an attributes object is used to create a thread, the values of the individual attributes determine the characteristics of the new thread. Attributes objects act as additional arguments to thread creation. Changing individual attributes does not affect any threads that were previously created using the attributes object.

You can use a single attributes object in multiple calls to `pthread_create`, from any thread. If multiple threads might change attributes in a shared attributes object, the application code must protect the integrity of the attributes object by using a mutex.



When you set the scheduling policy or scheduling parameters, or both, in an attributes object, you must disable scheduling inheritance if you want the scheduling attributes you set to be used at thread creation. To disable scheduling inheritance use the `pthread_attr_setinheritsched` routine, specifying the value `PTHREAD_EXPLICIT_SCHED` for the *inherit* argument, before calling `pthread_create`.

## Return Values

If an error condition occurs, the thread attributes object cannot be used and this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>attr</i> is invalid.
[ENOMEM]	Insufficient memory exists to initialize the thread attributes object.

## Associated Routines

pthread\_attr\_destroy  
 pthread\_attr\_setdetachstate  
 pthread\_attr\_setguardsize\_np  
 pthread\_attr\_setinheritsched  
 pthread\_attr\_setschedparam  
 pthread\_attr\_setschedpolicy  
 pthread\_attr\_setstacksize  
 pthread\_setguardsize\_np  
 pthread\_create



---

## pthread\_attr\_setdetachstate

Changes the *detachstate* attribute in the specified thread attributes object.

### Syntax

```
pthread_attr_setdetachstate(  
    attr,  
    detachstate);
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
detachstate	integer	read

### C Binding

```
#include <pthread.h>
```

```
int  
pthread_attr_setdetachstate (  
    pthread_attr_t *attr,  
    int detachstate);
```

### Arguments

#### attr

Thread attributes object to be modified.

#### detachstate

New value for the detachstate attribute. Valid values are as follows:

**PTHREAD\_CREATE\_JOINABLE** This is the default value. Threads are created in “undetached” state.

**PTHREAD\_CREATE\_DETACHED** The created thread is detached immediately, before it begins running.

### Description

This routine changes the *detachstate* attribute in the thread creation attributes. This attribute specifies whether the threads created using the specified thread attributes object are created in a detached state or not. A value of **PTHREAD\_CREATE\_JOINABLE** indicates the thread is not detached, and a value of **PTHREAD\_CREATE\_DETACHED** indicates the thread is detached. **PTHREAD\_CREATE\_JOINABLE** is the default value.

You cannot use the thread handle (the value of type `pthread_t` that is returned by `pthread_create`) for a detached thread. This means that you cannot cancel the thread with `pthread_cancel`. You also cannot use `pthread_join` to wait for the thread to complete or to retrieve the thread’s return status.

When a thread that has not been detached completes execution, DECthreads will retain the state of that thread to allow another thread to join with it. If the thread is detached before it completes, DECthreads is free to immediately reclaim the thread’s storage and resources. Failing to detach threads that have completed execution can result in wasting resources, so threads should be detached as soon



as the program is done with them. If there is no need to use the thread's handle after creation, create the thread initially detached.

## Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by the <i>detachstate</i> attribute is invalid.

## Associated Routines

pthread\_attr\_init  
pthread\_attr\_getdetachstate  
pthread\_create  
pthread\_join



## pthread\_attr\_setguardsize\_np

---

## pthread\_attr\_setguardsize\_np

Changes the guardsize attribute of the specified thread attributes object.

### Syntax

```
pthread_attr_setguardsize_np(  
    attr,  
    guardsize );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
guardsize	size_t	read

### C Binding

```
#include <pthread.h>  
  
int  
pthread_attr_setguardsize_np (  
    pthread_attr_t *attr,  
    size_t guardsize);
```

### Arguments

#### attr

Threads attributes object modified.

#### guardsize

New value for the guardsize attribute. The *guardsize* argument specifies the minimum size (in bytes) of the guard area for the stack of a thread.

### Description

This routine sets the minimum size (in bytes) of the guard area for the stack of a thread that is created using the attributes object specified by the *attr* argument.

A guard area helps to detect stack overflows by preventing memory access beyond the thread's stack. Large guard areas are necessary when threads might allocate large structures on the stack.

The default value is platform dependent, but will always be at least one "hardware protection unit" (at least one page).

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>attr</i> is invalid.



## Associated Routines

pthread\_attr\_init

pthread\_attr\_getguardsize\_np



## pthread\_attr\_setinheritsched

Changes the inherit scheduling attribute of the specified thread attributes object.

### Syntax

```
pthread_attr_setinheritsched(  
    attr,  
    inheritsched );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
inheritsched	integer	read

### C Binding

```
#include <pthread.h>  
  
int  
pthread_attr_setinheritsched (  
    pthread_attr_t *attr,  
    int inheritsched);
```

### Arguments

#### attr

Thread attributes object to be modified.

#### inheritsched

New value for the inherit scheduling attribute. Valid values are as follows:

**PTHREAD\_INHERIT\_SCHED**

The created thread inherits the scheduling policy and associated scheduling attributes of the thread calling pthread\_create. Any scheduling attributes in the attributes object specified by the pthread\_create attr argument are ignored during thread creation. This is the default value.

**PTHREAD\_EXPLICIT\_SCHED**

The scheduling policy and associated scheduling attributes of the created thread are set to the corresponding values from the attribute object specified by the pthread\_create attr argument.

### Description

This routine changes the inherit scheduling attribute of thread creation. The inherit scheduling attribute specifies whether threads created using the specified attributes object inherit the scheduling attributes of the creating thread, or use the scheduling attributes stored in the attributes object specified by the pthread\_create attr argument.



The first thread in an application has a scheduling policy of `SCHED_OTHER`. See the `pthread_attr_setschedparam` and `pthread_attr_setschedpolicy` routines for more information on valid priority values and valid scheduling policy values, respectively.

Inheriting scheduling attributes (instead of using the scheduling attributes stored in the attributes object) is useful when a thread is creating several helper threads—threads that are intended to work closely with the creating thread to cooperatively solve the same problem. For example, inherited scheduling attributes ensure that helper threads created in a sort routine execute with the same priority as the calling thread.

## Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	One or both of the values specified by <i>inherit</i> or by <i>attr</i> is invalid.
[ENOTSUP]	An attempt was made to set the attribute to an unsupported value.

## Associated Routines

pthread\_attr\_init  
 pthread\_attr\_getinheritsched  
 pthread\_attr\_setschedpolicy  
 pthread\_attr\_setschedparam  
 pthread\_create



## pthread\_attr\_setschedparam

---

## pthread\_attr\_setschedparam

Changes the values of the parameters associated with a scheduling policy of the specified thread attributes object.

### Syntax

```
pthread_attr_setschedparam(  
    attr,  
    param );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
param	struct sched_param	read

### C Binding

```
#include <pthread.h>  
  
int  
pthread_attr_setschedparam (  
    pthread_attr_t *attr,  
    const struct sched_param *param);
```

### Arguments

#### attr

Thread attributes object of the scheduling policy attribute whose parameters are to be set.

#### param

A structure containing new values for scheduling parameters associated with the scheduling policy attribute defined for the specified thread attributes object.

---

#### Note

DECthreads provides only the sched\_priority scheduling parameter. It allows you to specify the scheduling priority. See below for information about this scheduling parameter.

---

### Description

This routine sets the scheduling parameters associated with the scheduling policy attribute for the thread attributes object specified by the *attr* argument.

#### Scheduling Priority

Use the sched\_priority scheduling parameter to set a thread's execution priority. The effect of the scheduling priority you assign depends on the scheduling policy specified for the attributes object specified by the *attr* argument.



By default, a created thread inherits the priority of the thread calling `pthread_create`. To specify a priority using this routine, scheduling inheritance must be disabled at the time the thread is created. Call `pthread_attr_setinheritsched` and specify the value `PTHREAD_EXPLICIT_SCHED` for the *inherit* argument before calling `pthread_create`.

An application specifies priority only to express the urgency of executing the thread relative to other threads. **DO NOT USE PRIORITY TO CONTROL MUTUAL EXCLUSION WHEN ACCESSING SHARED DATA.** With a sufficient number of processors executing, all ready threads, regardless of priority, execute simultaneously.

Valid values of the `sched_priority` scheduling parameter depend on the chosen policy and fall within one of the following ranges:

Policy	Low	High
<code>SCHED_FIFO</code>	<code>PRI_FIFO_MIN</code>	<code>PRI_FIFO_MAX</code>
<code>SCHED_RR</code>	<code>PRI_RR_MIN</code>	<code>PRI_RR_MAX</code>
<code>SCHED_OTHER</code>	<code>PRI_OTHER_MIN</code>	<code>PRI_OTHER_MAX</code>
<code>SCHED_FG_NP</code>	<code>PRI_FG_MIN_NP</code>	<code>PRI_FG_MAX_NP</code>
<code>SCHED_BG_NP</code>	<code>PRI_BG_MIN_NP</code>	<code>PRI_BG_MAX_NP</code>

The default priority is the midpoint between `PRI_OTHER_MIN` and `PRI_OTHER_MAX` for the `SCHED_OTHER` policy. (Section 2.7 describes how to specify priorities between the minimum and maximum values.)

## Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>param</i> is invalid.
[ENOTSUP]	An attempt was made to set the attribute to an unsupported value.

## Associated Routines

`pthread_attr_init`  
`pthread_attr_getschedparam`  
`pthread_attr_setinheritsched`  
`pthread_attr_setschedpolicy`  
`pthread_create`



## pthread\_attr\_setschedpolicy

Changes the scheduling policy of the specified thread attributes object.

### Syntax

```
pthread_attr_setschedpolicy(
    attr,
    policy );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
policy	integer	read

### C Binding

```
#include <pthread.h>

int
pthread_attr_setschedpolicy (
    pthread_attr_t *attr,
    int policy);
```

### Arguments

#### attr

Thread attributes object to be modified.

#### policy

New value for the scheduling policy attribute. Valid values are as follows:

```
SCHED_BG_NP
SCHED_FG_NP (also known as SCHED_OTHER)
SCHED_FIFO
SCHED_RR
```

SCHED\_OTHER is the default value. See Section 2.2.3.2 for a description of the scheduling policies.

### Description

This routine sets the scheduling policy of a thread that is created using the attributes object specified by the *attr* argument. The default value of the scheduling attribute is SCHED\_OTHER.

By default, a created thread inherits the priority of the thread calling `pthread_create`. To specify a priority using this routine, scheduling inheritance must be disabled at the time the thread is created. Call `pthread_attr_setinheritsched` and specify the value `PTHREAD_EXPLICIT_SCHED` for the *inherit* argument before calling `pthread_create`.

Never attempt to use scheduling as a mechanism for synchronization. (Refer to Chapter 1 and Chapter 2.)



## Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>policy</i> is invalid.

## Associated Routines

pthread\_attr\_init  
 pthread\_attr\_getschedpolicy  
 pthread\_attr\_setinheritsched  
 pthread\_attr\_setschedparam  
 pthread\_create



## pthread\_attr\_setstacksize

Changes the *stacksize* attribute in the specified thread attributes object.

### Syntax

```
pthread_attr_setstacksize(
    attr,
    stacksize );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	write
stacksize	size_t	read

### C Binding

```
#include <pthread.h>

int
pthread_attr_setstacksize (
    pthread_attr_t *attr,
    size_t stacksize);
```

### Arguments

#### attr

Threads attributes object modified.

#### stacksize

New value for the stacksize attribute. The *stacksize* argument must be greater than or equal to PTHREAD\_STACK\_MIN. PTHREAD\_STACK\_MIN specifies the minimum size (in bytes) of stack needed for a thread.

### Description

This routine sets the thread creation stacksize attribute in the *attr* object. Use this routine to adjust the size of the writable area of the stack for threads that will be created.

A thread's stack is fixed at the time of thread creation. Only the initial thread can dynamically extend its stack.

Many compilers do not check for stack overflow. Ensure that your thread stack is large enough for anything that you call from the thread.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.



Return	Description
[EINVAL]	The value specified by <i>attr</i> is invalid or the value specified by <i>stacksize</i> is less than {PTHREAD_STACK_MIN} or exceeds a system-imposed limit.

## Associated Routines

pthread\_attr\_init  
 pthread\_attr\_getstacksize  
 pthread\_create



## pthread\_cancel

Allows a thread to request that it or another thread terminate execution.

### Syntax

```
pthread_cancel(  
    thread );
```

Argument	Data Type	Access
thread	opaque pthread_t	read

### C Binding

```
#include <pthread.h>

int
pthread_cancel (
    pthread_t  thread);
```

### Arguments

**thread**

Thread that receives a cancel request.

### Description

This routine sends a cancel to the specified thread. A cancel is a mechanism by which a calling thread requests the specified thread to terminate as quickly as possible. Issuing a cancel does not guarantee that the canceled thread will receive or handle the cancel. When the cancellation is acted on, all active cleanup handlers for *thread* are called. When the last cleanup handler returns, the thread-specific data destructor functions shall be called for each thread-specific data key with a destructor and for which the thread has a non-NULL value. Finally, the *thread* is terminated.

The cancellation processing in the target thread runs asynchronously with respect to the calling thread returning from `pthread_cancel()`. The target thread cancelability state and type determine when or if the cancellation takes place:

1. The canceled thread can delay cancellation during critical operations by setting its cancelability state to `PTHREAD_CANCEL_DISABLE`.
2. Because of communication delays, the calling thread can only rely on the fact that a cancel will eventually become pending in the designated thread (provided that the thread does not terminate beforehand).
3. The calling thread has no guarantee that a pending cancel will be delivered because delivery is controlled by the designated thread.

When a cancel is delivered to a thread, termination processing is similar to `pthread_exit`. For more information about thread termination, see the Thread Termination section of `pthread_create`.

This routine is preferred in implementing an Ada abort statement and any other language- or software-defined construct for requesting thread cancellation.



The results of this routine are unpredictable if the value specified in *thread* refers to a thread that does not currently exist.

## Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The specified thread is invalid.
[ESRCH]	<i>thread</i> does not specify an existing thread.

## Associated Routines

pthread\_cleanup\_pop  
pthread\_cleanup\_push  
pthread\_create  
pthread\_exit  
pthread\_join  
pthread\_setcancelstate  
pthread\_setcanceltype  
pthread\_testcancel



## pthread\_cleanup\_pop

---

## pthread\_cleanup\_pop

Removes the cleanup handler and optionally executes it.

### Syntax

```
pthread_cleanup_pop(  
    execute );
```

Argument	Data Type	Access
<i>execute</i>	integer	read

### C Binding

```
#include <pthread.h>  
  
void  
pthread_cleanup_pop(  
    int execute);
```

### Arguments

#### **execute**

Integer that specifies whether the cleanup routine specified in the matching call to `pthread_cleanup_push` is executed. A nonzero value will cause the routine to be executed. This routine can be used to clean up from a block of code whether exited by cancellation or normal completion.

### Description

This routine removes the cleanup routine and executes it if the value specified in *execute* is nonzero.

This routine and `pthread_cleanup_push` are implemented as macros and must appear as statements and in pairs within the same lexical scope. You can think of the `pthread_cleanup_push` macro as expanding to a string whose first character is a left brace (`{`) and `pthread_cleanup_pop` expanding to a string containing the corresponding right brace (`}`).

### Return Values

None

### Associated Routines

```
pthread_cancel  
pthread_cleanup_push  
pthread_create  
pthread_exit
```



## pthread\_cleanup\_push

Establishes a cleanup handler to be executed when the thread exits or is canceled.

### Syntax

```
pthread_cleanup_push(
    routine,
    arg );
```

Argument	Data Type	Access
routine	procedure	read
arg	user_arg	read

### C Binding

```
#include <pthread.h>

void
pthread_cleanup_push(
    void (*routine)(void *),
    void *arg);
```

### Arguments

#### routine

Routine executed as the cleanup handler.

#### arg

Argument executed with the cleanup routine.

### Description

This routine pushes the specified routine onto the calling thread's cleanup stack. The cleanup routine is popped from the stack and executed with the *arg* argument when any of the following actions occur:

- The thread calls `pthread_exit`.
- The thread is canceled.
- An exception is raised (while the cleanup routine is in effect) and results in an unwind through the scope of the `pthread_cleanup_push` and `pthread_cleanup_pop` pair.
- The thread calls `pthread_cleanup_pop` and specifies a nonzero value for the *execute* argument.

This routine and `pthread_cleanup_pop` are implemented as macros and must appear as statements and in pairs within the same lexical scope. You can think of the `pthread_cleanup_push` macro as expanding to a string whose first character is a left brace (`{`) and `pthread_cleanup_pop` as expanding to a string containing the corresponding right brace (`}`).



## Return Values

None

## Associated Routines

pthread\_cancel  
pthread\_cleanup\_pop  
pthread\_create  
pthread\_exit  
pthread\_testcancel



## pthread\_cond\_broadcast

Wakes all threads that are waiting on a condition variable.

### Syntax

```
pthread_cond_broadcast(  
    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify

### C Binding

```
#include <pthread.h>

int
pthread_cond_broadcast (
    pthread_cond_t *cond);
```

### Arguments

**cond**  
Condition variable broadcast.

### Description

This routine unblocks all threads waiting on the specified condition variable *cond*. Calling this routine implies that data guarded by the associated mutex has changed so that it might be possible for one or more waiting threads to proceed. The threads that are unblocked shall contend for the mutex according to the scheduling policy (if applicable).

If only one of the threads waiting on a condition variable may be able to proceed, but any single thread can proceed, then use `pthread_cond_signal` instead.

Whether the associated mutex is locked or unlocked, you can still call this routine. However, if predictable scheduling behavior is required, then that mutex should then be locked by the thread calling the `pthread_cond_broadcast` routine.

If no threads are waiting on the specified condition variable then this routine takes no action. The broadcast is not preserved for the next condition variable wait.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>cond</i> is invalid.



## Associated Routines

pthread\_cond\_destroy  
pthread\_cond\_init  
pthread\_cond\_signal  
pthread\_cond\_timedwait  
pthread\_cond\_wait



## pthread\_cond\_destroy

Destroys a condition variable.

### Syntax

```
pthread_cond_destroy(
    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	write

### C Binding

```
#include <pthread.h>

int
pthread_cond_destroy (
    pthread_cond_t  *cond);
```

### Arguments

**cond**  
Condition variable to destroy.

### Description

This routine destroys the condition variable specified by *cond*. This effectively uninitializes the condition variable. You call this routine when a condition variable will no longer be referenced. Destroying a condition variable may allow DECThreads to reclaim internal memory associated with the condition variable.

It is safe to destroy an initialized condition variable upon which no threads are currently blocked. Attempting to destroy a condition variable upon which other threads are blocked results in unpredictable behavior.

The results of this routine are unpredictable if the condition variable specified in *cond* does not exist or is not initialized.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>cond</i> is invalid.
[EBUSY]	The object being referenced by <i>cond</i> is being referenced by another thread that is currently executing a <code>pthread_cond_wait</code> or <code>pthread_cond_timedwait</code> on the condition variable specified in <i>cond</i> .



## Associated Routines

pthread\_cond\_broadcast  
pthread\_cond\_init  
pthread\_cond\_signal  
pthread\_cond\_timedwait  
pthread\_cond\_wait

Argument	Default Value	Return Value
cond	pthread_cond_t	0 on success, non-zero on error

Description
<p>This routine destroys the condition variable <i>cond</i>. It is not necessary to call this routine if the condition variable was created with <code>pthread_cond_t</code> and not <code>pthread_cond_t</code>. The condition variable <i>cond</i> must not be held by any thread when this routine is called. The condition variable <i>cond</i> must not be held by any thread when this routine is called. The condition variable <i>cond</i> must not be held by any thread when this routine is called.</p>

Portability Notes
<p>Linux: Condition variables are implemented using the <code>pthread_cond_t</code> type. The condition variable <i>cond</i> must not be held by any thread when this routine is called.</p>



## pthread\_cond\_init

Initializes a condition variable.

### Syntax

```
pthread_cond_init(
    cond,
    attr);
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	write
attr	opaque pthread_condattr_t	read

### C Binding

```
#include <pthread.h>

int
pthread_cond_init (
    pthread_cond_t *cond,
    const pthread_condattr_t *attr);
```

### Arguments

#### cond

Condition variable to be initialized.

#### attr

Condition variable attributes object that defines the characteristics of the condition variable initialized.

### Description

This routine initializes a condition variable (*cond*) with attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes are used.

A condition variable is a synchronization object used in conjunction with a mutex. A mutex controls access to shared data; a condition variable allows threads to wait for that data to enter a defined state.

Condition variables are not owned by a particular thread. Any associated storage is not automatically deallocated when the creating thread terminates.

The macro PTHREAD\_COND\_INITIALIZER can be used to initialize statically allocated condition variables to the default condition variable attributes. To call this macro, enter:

```
pthread_cond_t condition = PTHREAD_COND_INITIALIZER
```

When statically initialized, a condition variable should not also be using pthread\_cond\_init. Also, a statically initialized condition variable need not be destroyed using pthread\_cond\_destroy.



## pthread\_cond\_init

Under certain circumstances it may be impossible to wait upon a statically initialized condition variable when the process virtual address space (or some other memory limit) is nearly exhausted. In such a case `pthread_cond_wait` or `pthread_cond_timedwait` may return `ENOMEM`. You can avoid this possibility by initializing critical condition variables with `pthread_cond_init`.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error, the condition variable is not initialized, and the contents of *cond* are undefined. Possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacks the necessary resources to initialize another condition variable, or The system-imposed limit on the total number of condition variables under execution by a single user is exceeded.
[EBUSY]	The implementation has detected an attempt to reinitialize the object referenced by <i>cond</i> , a previously initialized, but not yet destroyed condition variable.
[EINVAL]	The value specified by <i>attr</i> is invalid.
[ENOMEM]	Insufficient memory exists to initialize the condition variable.

### Associated Routines

`pthread_cond_broadcast`  
`pthread_cond_destroy`  
`pthread_cond_signal`  
`pthread_cond_timedwait`  
`pthread_cond_wait`



---

## pthread\_cond\_signal

Wakes at least one thread that is waiting on a condition variable.

### Syntax

```
pthread_cond_signal(  
    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify

### C Binding

```
#include <pthread.h>  
  
int  
pthread_cond_signal (  
    pthread_cond_t    *cond);
```

### Arguments

**cond**  
Condition variable signaled.

### Description

This routine unblocks at least one thread waiting on the specified condition variable *cond*. Calling this routine implies that data guarded by the associated mutex has changed so that it might be possible for one of the waiting threads to proceed. In general, only one will be released.

If no threads are waiting on the specified condition variable then this routine takes no action. The signal is not preserved for the next condition variable wait.

This routine should be called when any thread waiting on the specified condition variable might find its predicate true, but only one thread should proceed. If more than one thread can proceed, or if any thread would not be able to proceed, then you must use `pthread_cond_broadcast`.

The scheduling policy determines which thread is awakened. For policies `SCHED_FIFO` and `SCHED_RR`, a blocked thread is chosen in priority order, using first-in/first-out (FIFO) within priorities.

You can call this routine even when the associated mutex is locked. However, if predictable scheduling behavior is required, then that mutex should be locked by the thread calling `pthread_cond_signal`.



## pthread\_cond\_signal

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>cond</i> is invalid.

### Associated Routines

pthread\_cond\_broadcast  
pthread\_cond\_destroy  
pthread\_cond\_init  
pthread\_cond\_timedwait  
pthread\_cond\_wait



---

## pthread\_cond\_signal\_int\_np

Wakes one thread that is waiting on a condition variable (called from interrupt level only).

### Syntax

```
pthread_cond_signal_int_np(
    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify

### C Binding

```
#include <pthread.h>

int
pthread_cond_signal_int_np(
    pthread_cond_t *cond);
```

### Arguments

**cond**  
Condition variable signaled.

### Description

This routine wakes one thread waiting on a condition variable. It can only be called from a software interrupt handler routine (Digital UNIX signal handler or OpenVMS AST). Calling this routine implies that it might be possible for a single waiting thread to proceed.

The scheduling policies of the waiting threads determine which thread is awakened. For policies `SCHED_FIFO` and `SCHED_RR`, a blocked thread is chosen in priority order, using first-in/first-out (FIFO) within priorities.

This routine does not cause a thread blocked on a condition variable to resume execution immediately. A thread resumes execution at some time after the interrupt handler returns.

You can call this routine regardless of whether the associated mutex is locked. Never try to lock a mutex from an interrupt handler.

---

#### Note

---

This routine allows you to signal a thread from a software interrupt handler. Do not call this routine from noninterrupt code. If you want to signal a thread from the normal noninterrupt level, use `pthread_cond_signal`.

---



## pthread\_cond\_signal\_int\_np

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>cond</i> is invalid.

### Associated Routines

pthread\_cond\_wait  
pthread\_cond\_timedwait  
pthread\_cond\_signal  
pthread\_cond\_broadcast



## pthread\_cond\_timedwait

Causes a thread to wait for a condition variable to be signaled or broadcasted such that it will awake after a specified period of time.

### Syntax

```
pthread_cond_timedwait(
    cond,
    mutex,
    abstime );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify
mutex	opaque pthread_mutex_t	modify
abstime	structure timespec	read

### C Binding

```
#include <pthread.h>

int
pthread_cond_timedwait (
    pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct timespec *abstime);
```

### Arguments

#### cond

Condition variable waited on.

#### mutex

Mutex associated with the condition variable specified in *cond*.

#### abstime

Absolute time at which the wait expires, if the condition has not been signaled or broadcasted. See the `pthread_get_expiration_np` routine, which is used to obtain a value for this argument.

The *abstime* argument is specified in Universal Coordinated Time (UTC). In the UTC-based model, time is represented as seconds since the Epoch. The Epoch is defined as the time 0 hours, 0 minutes, 0 seconds, January 1st, 1970 UTC. Seconds since the Epoch is a value interpreted as the number of seconds between a specified time and the Epoch.

### Description

This routine causes a thread to wait until one of the following occurs:

- The specified condition variable is signaled or broadcasted.
- The current system clock time is greater than or equal to the time specified by the *abstime* argument.



## pthread\_cond\_timedwait

This routine is identical to `pthread_cond_wait` except that this routine can return before a condition variable is signaled or broadcasted; specifically, when a specified time expires. For more information, see the `pthread_cond_wait` description.

This routine atomically releases the mutex and causes the calling thread to wait on the condition. The atomicity is important, because it means the thread cannot miss a wakeup while the mutex is unlocked. When the timer expires or when the wait is satisfied as a result of some thread calling `pthread_cond_signal` or `pthread_cond_broadcast`, the mutex is reacquired before returning to the caller.

If the current time equals or exceeds the expiration time, this routine returns immediately, without releasing the mutex or causing the current thread to wait. Your code should check the return status whenever this routine returns and take the appropriate action. Otherwise, waiting on the condition variable can become a nonblocking loop.

Call this routine after you lock the mutex specified in *mutex*. The results of this routine are unpredictable if this routine is called without first locking the mutex.

### Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>cond</i> , <i>mutex</i> , or <i>abstime</i> is invalid, or: Different mutexes are supplied for concurrent <code>pthread_cond_timedwait</code> operations or <code>pthread_cond_wait</code> operations on the same condition variable, or: The mutex was not owned by the current thread at the time of the call.
[ETIMEDOUT]	The time specified by <i>abstime</i> expired.
[ENOMEM]	DECthreads cannot acquire memory needed to block using a statically initialized condition variable.

### Associated Routines

`pthread_cond_broadcast`  
`pthread_cond_destroy`  
`pthread_cond_init`  
`pthread_cond_signal`  
`pthread_cond_wait`  
`pthread_get_expiration_np`



---

## pthread\_cond\_wait

Causes a thread to wait for a condition variable to be signaled or broadcasted.

### Syntax

```
pthread_cond_wait(
    cond,
    mutex );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify
mutex	opaque pthread_mutex_t	modify

### C Binding

```
#include <pthread.h>

int
pthread_cond_wait (
    pthread_cond_t *cond,
    pthread_mutex_t *mutex);
```

### Arguments

#### cond

Condition variable waited on.

#### mutex

Mutex associated with the condition variable specified in *cond*.

### Description

This routine causes a thread to wait for a condition variable to be signaled or broadcasted. Each condition corresponds to one or more Boolean relations (predicates) based on shared data. The calling thread waits for the data to reach a particular state for the predicate to become true. However, the return from this routine does not imply anything about the value of the predicate and it should be reevaluated upon return. Condition variables are discussed in Chapter 2 and Chapter 3.

Call this routine after you have locked the mutex specified in *mutex*. The results of this routine are unpredictable if this routine is called without first locking the mutex.

This routine atomically releases the mutex and causes the calling thread to wait on the condition. The atomicity is important, because it means the thread cannot miss a wakeup while the mutex is unlocked. When the wait is satisfied as a result of some thread calling `pthread_cond_signal` or `pthread_cond_broadcast`, the mutex is reacquired before returning to the caller.

A thread that changes the state of storage protected by the mutex in such a way that a predicate associated with a condition variable might now be true must call either `pthread_cond_signal` or `pthread_cond_broadcast` for that condition



## pthread\_cond\_wait

variable. If neither call is made, any thread waiting on the condition variable continues to wait.

This routine might (with low probability) return when the condition variable has not been signaled or broadcasted. When this occurs, the mutex is reacquired before the routine returns. To handle this type of situation, enclose this routine in a loop that checks the predicate. The loop provides documentation of your intent and protects against these spurious wakeups allowing correct behavior even if another thread consumes the desired state before the awakened thread runs.

It is illegal for threads to wait on the same condition variable specifying different mutexes.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>cond</i> or <i>mutex</i> is invalid, or: Different mutexes are supplied for concurrent <code>pthread_cond_wait</code> or <code>pthread_cond_timedwait</code> operations on the same condition variable, or The mutex was not owned by the current thread at the time of the call.
[ENOMEM]	DECthreads cannot acquire memory needed to block using a statically initialized condition variable.

### Associated Routines

`pthread_cond_broadcast`  
`pthread_cond_destroy`  
`pthread_cond_init`  
`pthread_cond_signal`  
`pthread_cond_timedwait`



## pthread\_condattr\_destroy

Destroys a condition variable attributes object.

### Syntax

```
pthread_condattr_destroy(
    attr );
```

Argument	Data Type	Access
attr	opaque pthread_condattr_t	write

### C Binding

```
#include <pthread.h>

int
pthread_condattr_destroy (
    pthread_condattr_t  *attr);
```

### Arguments

**attr**  
Condition variable attributes object to be destroyed.

### Description

This routine destroys a condition variable attributes object (the object becomes uninitialized).

Condition variables that were created using this attributes object are not affected by the deletion of the condition variable attributes object.

After calling this routine, the results of using *attr* in a call to any routine (other than pthread\_cond\_attr\_init) are unpredictable.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The attributes object specified by <i>attr</i> is invalid.

### Associated Routines

pthread\_condattr\_init



## pthread\_condattr\_init

Initializes a condition variable attributes object that can be used to specify the attributes of condition variables when they are created.

### Syntax

```
pthread_condattr_init(
    attr );
```

Argument	Data Type	Access
attr	opaque pthread_condattr_t	write

### C Binding

```
#include <pthread.h>

int
pthread_condattr_init (
    pthread_condattr_t  *attr);
```

### Arguments

**attr**  
Condition variable attributes object to initialize.

### Description

This routine initializes the condition variable attributes object (*attr*) that is used to specify the attributes of condition variables when they are initialized. The condition variable attributes object is initialized with the default attribute values.

When a condition variable attributes object is used to initialize a condition variable, the values of the individual attributes determine the characteristics of the new object. Attributes objects act like additional arguments to object initialization. Changing individual attributes does not affect objects that were previously initialized using the attributes object.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[ENOMEM]	Insufficient memory exists to initialize the condition variable attributes object.



## Associated Routines

pthread\_condattr\_destroy  
pthread\_cond\_init



## pthread\_create

Creates a thread.

### Syntax

```
pthread_create(
    thread,
    attr,
    start_routine,
    arg );
```

Argument	Data Type	Access
thread	opaque pthread_t	write
attr	opaque pthread_attr_t	read
start_routine	procedure	read
arg	user_arg	read

### C Binding

```
#include <pthread.h>

int
pthread_create (
    pthread_t *thread,
    const pthread_attr_t *attr,
    void * (*start_routine) (void *),
    void *arg);
```

### Arguments

**thread**  
Thread object created.

**attr**  
Thread attributes object that defines the characteristics of the thread being created. If you specify NULL, default attributes are used.

**start\_routine**  
Function executed as the new thread's start routine.

**arg**  
Address value copied and passed to the thread's start routine.

### Description

This routine creates a thread. A **thread** is a single, sequential flow of control within a program. It is the active execution of a designated routine, including any nested routine invocations.

Successful execution of this routine includes the following actions:

- An internal **thread object** is created to describe and control the thread. The thread object includes a Thread Environment Block (TEB) that programs can use (with care).



- The *thread* argument receives an identifier for the new thread.
- An executable thread is created with attributes specified by the *attr* argument (or with default attributes if NULL is specified).

### Thread Creation

A thread is created in the ready state to begin executing the function specified by the *start\_routine* argument. The new thread may execute immediately. The newly created thread may preempt its creator, depending on scheduling policy and priority. The thread identifier (`pthread_t`) specified in `pthread_create` will be written before the new thread executes.

The new thread's scheduling policy and priority are, by default, inherited from the creating thread—the scheduling policy and priority set in the attributes object are ignored. To create a thread using the scheduling policy and priority set in the attributes object, you must first disable the inherit scheduling attribute by calling `pthread_attr_setinheritsched` before calling `pthread_create`.

The thread exists until it is both terminated and detached. A thread is detached when created if the detach-state attribute is set to `PTHREAD_CREATE_DETACHED`. It is also detached after any thread returns successfully from calling `pthread_detach` or `pthread_join` for the thread. Termination is explained in the next section (see Thread Termination).

The caller of `pthread_create` can synchronize with the newly created thread through the use of the `pthread_join` routine, or any other mutexes or condition variables they agree to use.

On Digital UNIX, the signal state of the new thread is initialized as follows:

1. The signal mask is inherited from the creating thread.
2. The set of signals pending for the new thread is empty.

If `pthread_create` fails, no new thread is created and the contents of the location referenced by *thread* are undefined.

### Thread Termination

A thread terminates when one of the following events occurs:

- The thread returns from its start routine.
- An exception is raised (or re-raised) in the thread during execution and the exception is not handled.
- The thread calls the `pthread_exit` routine.
- The thread is canceled.

The following actions are performed when a thread terminates:

- If the thread has been canceled, a return value of `PTHREAD_CANCELED` is copied into the thread object. If the thread terminated by returning from its start routine or calling `pthread_exit`, the return value is copied into the thread object. The return value can be retrieved by other threads by calling the `pthread_join` routine.

If the start routine returns normally and the start routine is a procedure that does not return a value, then the result obtained by `pthread_join` is unpredictable.



## pthread\_create

- If the termination results from a cancellation, an unhandled exception, or a call to `pthread_exit`, each cleanup handler that has been declared by `pthread_cleanup_push` and not yet removed by `pthread_cleanup_pop` is called. Cleanup handlers are called in order, starting with the most recently pushed handler.
- For each thread-specific data key for which the thread has a non-NULL value, the destructor routine (if available) is called. Destructors are called in unspecified order. Before each destructor is called, the thread's value for the corresponding key is set to NULL. This step repeats until all thread-specific data values in the thread are NULL, or for up to four iterations (`PTHREAD_DESTRUCTOR_ITERATIONS`).
- The thread waiting in a call to `pthread_join`, if any, is awakened.
- The thread object is marked to indicate that it is no longer needed by the thread itself. A check is made to determine if the thread is already detached. If it is detached, then the thread object is released.

### Return Values

If an error condition occurs, no thread is created, the contents of *thread* are undefined, and this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacks the necessary resources to create another thread, or the system-imposed limit on the total number of threads under execution by a single user is exceeded.
[EINVAL]	The value specified by <i>attr</i> is invalid.
[ENOMEM]	Insufficient enough memory exists to create a thread.
[EPERM]	The caller does not have the appropriate permission to create a thread with the specified attributes.

### Associated Routines

`pthread_atfork`  
`pthread_attr_destroy`  
`pthread_attr_init`  
`pthread_attr_setdetachstate`  
`pthread_attr_setinheritsched`  
`pthread_attr_setschedparam`  
`pthread_attr_setschedpolicy`  
`pthread_attr_setstacksize`  
`pthread_cancel`  
`pthread_detach`  
`pthread_exit`  
`pthread_join`



---

## pthread\_debug

Invokes the DECthreads internal debugger.

### Syntax

```
pthread_debug( );
```

### C Binding

```
#include <pthread.h>
```

```
void
```

```
pthread_debug (void);
```

### Arguments

None

### Description

This routine invokes the DECthreads internal debugger as a callable function. It takes no arguments and does not return a value. It enters the internal debugger parsing loop. Type `exit` to return to the program.

To pass a list of debugging commands to DECthreads, call `pthread_debug_cmd`.

For more information, see Appendix D.

### Return Values

None

### Associated Routines

`pthread_debug_cmd`



## pthread\_debug\_cmd

Passes a list of pthread\_debug commands to DECthreads.

### Syntax

```
pthread_debug_cmd(
    cmd );
```

Argument	Data Type	Access
cmd	character string	read

### C Binding

```
#include <pthread.h>
pthreadDbgStatus_t
pthread_debug_cmd (
    char *cmd);
```

### Arguments

#### cmd

pthread\_debug command string to DECthreads debug. Null terminated string, commands separated by semicolons.

### Description

This routine passes a list of debugging commands to DECthreads. Each command (separated by a semicolon) is executed in sequence. Any output is written to standard output. This routine returns the status of the last specified operation in the command string when the final command (or Exit command) is executed.

For a list of pthread\_debug commands, see Appendix D.

The following are two examples of listing commands in a call to this routine:

```
pthread_debug_cmd("thread -b; mu -lq; cond -wq");
pthread_debug_cmd("att");
```

If you want to invoke the debugger for interactive commands, call pthread\_debug.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Command successful
[PTHREAD_DBG_QUIT] (1)	Last command was quit or exit
[PTHREAD_DBG_NONESEL] (2)	No objects selected (for example, thread -br)
[PTHREAD_DBG_SUCCESSPEND] (3)	Alternate success



Return	Description
[PTHREAD_DBG_NOPRIV] (-1)	No privilege for command
[PTHREAD_DBG_INVPARAM] (-2)	Invalid parameter on command
[PTHREAD_DBG_INVSEQ] (-3)	Invalid object sequence number given
[PTHREAD_DBG_INCONSTATE] (-4)	Inconsistent state for operation
[PTHREAD_DBG_CORRUPT] (-5)	Unable to complete due to internal corruption
[PTHREAD_DBG_INVOPTION] (-6)	Invalid command options
[PTHREAD_DBG_NOARG] (-7)	Missing command argument
[PTHREAD_DBG_INVADDR] (-8)	Invalid address
[PTHREAD_DBG_INVCMD] (-9)	Invalid command
[PTHREAD_DBG_NULLCMD] (-10)	No command given
[PTHREAD_DBG_CONFLICT] (-11)	Conflicting options
[PTHREAD_DBG_UNIMPL] (-12)	Unimplemented feature

## Associated Routines

pthread\_debug



---

## pthread\_delay\_np

Causes a thread to delay execution.

### Syntax

```
pthread_delay_np(
    interval );
```

Argument	Data Type	Access
interval	struct timespec	read

### C Binding

```
#include <pthread.h>
int
pthread_delay_np (
    const struct timespec  *interval);
```

### Arguments

#### interval

Number of seconds and nanoseconds to delay execution. The value specified for each must be greater than or equal to zero.

### Description

This routine causes a thread to delay execution for a specific period of time. This period ends at the current time plus the specified interval. The routine will not return before the end of the period is reached, but may return an arbitrary amount of time after the period has gone by. This is due to system load, thread priorities, and system timer granularity.

Specifying an interval of 0 seconds and 0 nanoseconds is allowed and can be used to force the thread to give up the processor or to deliver a pending cancel.

The struct timespec structure contains the following two fields:

- tv\_sec is an integer number of seconds
- tv\_nsec is an integer number of nanoseconds

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>interval</i> is invalid.



## pthread\_detach

Marks a thread object for deletion.

### Syntax

```
pthread_detach(
    thread );
```

Argument	Data Type	Access
thread	opaque pthread_t	read

### C Binding

```
#include <pthread.h>

int
pthread_detach (
    pthread_t  thread);
```

### Arguments

**thread**  
Thread object being marked for deletion.

### Description

A call to this routine indicates that storage for the specified thread can be reclaimed when the thread terminates. This includes storage for the *thread* argument's return value, as well as the thread object. If *thread* has not terminated when this routine is called, this routine does not cause it to terminate.

When a thread object is no longer referenced, call this routine.

The results of this routine are unpredictable if the value of *thread* refers to a thread object that does not exist.

A thread can be created "pre-detached" using the detach-state attribute. The `pthread_join` function also detaches the target thread when `pthread_join` returns successfully.

### Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>thread</i> does not refer to a joinable thread.
[ESRCH]	The value specified by <i>thread</i> cannot be found.



## Associated Routines

pthread\_cancel  
pthread\_create  
pthread\_exit  
pthread\_join



## pthread\_equal

Compares one thread identifier to another thread identifier.

### Syntax

```
pthread_equal(  
    t1,  
    t2);
```

Argument	Data Type	Access
t1	opaque pthread_t	read
t2	opaque pthread_t	read

### C Binding

```
#include <pthread.h>  
  
int  
pthread_equal (  
    pthread_t  t1,  
    pthread_t  t2);
```

### Arguments

**t1**

The first thread identifier to be compared.

**t2**

The second thread identifier to be compared.

### Description

This routine compares one thread identifier to another thread identifier.

If either *t1* or *t2* are not valid thread IDs, the behavior is undefined.

### Return Values

Possible return values are as follows:

Return	Description
0	Values of <i>t1</i> and <i>t2</i> do not designate the same object.
Non-zero	Values of <i>t1</i> and <i>t2</i> designate the same object.



## pthread\_exit

Terminates the calling thread.

### Syntax

```
pthread_exit(
    value_ptr );
```

Argument	Data Type	Access
value_ptr	void *	read

### C Binding

```
#include <pthread.h>

void
pthread_exit (
    void    *value_ptr);
```

### Arguments

#### value\_ptr

Value copied and returned to the caller of pthread\_join.

Note that void \* is used as a universal datatype, not as a pointer. DECThreads treats the *value\_ptr* as a value and stores it to be returned by pthread\_join.

### Description

This routine terminates the calling thread and makes a status value (*value\_ptr*) available to any thread that calls pthread\_join and specifies the terminating thread.

Any cancellation cleanup handlers that have been pushed and not yet popped from the stack, are popped in the reverse order that they were pushed and then executed. After all cancellation cleanup handlers have been executed, appropriate destructor functions shall be called in an unspecified order if the thread has any thread-specific data. Thread termination does not release any application visible process resources, including, but not limited to mutexes and file descriptors, nor does it perform any process level cleanup actions, including, but not limited to calling any atexit routine that may exist.

An implicit call to pthread\_exit is issued when a thread returns from the start routine that was used to create it. The function's return value serves as the thread's exit status. The process exits when the last running thread calls pthread\_exit.

After a thread has terminated, the result of access to local (auto) variables of the thread is undefined. So, references to local variables of the existing thread should not be used for the value\_ptr parameter value of the pthread\_exit routine.



**Return Values**

None

**Associated Routines**

pthread\_cancel  
 pthread\_create  
 pthread\_detach  
 pthread\_join



---

## pthread\_get\_expiration\_np

Obtains a value representing a desired expiration time.

### Syntax

```
pthread_get_expiration_np(
    delta,
    abstime );
```

Argument	Data Type	Access
delta	struct timespec	read
abstime	struct timespec	write

### C Binding

```
#include <pthread.h>

int
pthread_get_expiration_np (
    const struct timespec *delta,
    struct timespec *abstime);
```

### Arguments

#### delta

Number of seconds and nanoseconds to add to the current system time. (The result is the time in the future.) This result will be place in *abstime*.

#### abstime

Value representing the absolute expiration time. The absolute expiration time is obtained by adding *delta* to the current system time. The resulting *abstime* is in Universal Coordinated Time (UTC). This value should be passed to the `pthread_cond_timedwait` routine.

### Description

This routine adds a specified interval to the current absolute system time and returns a new absolute time. This new absolute time is used as the expiration time in a call to `pthread_cond_timedwait`.

The struct timespec structure contains the following two fields:

- tv.sec is an integer number of seconds
- tv.nsec is an integer number of nanoseconds



## Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>delta</i> is invalid.

## Associated Routines

pthread\_cond\_timedwait



## pthread\_getschedparam

Obtains the current scheduling policy and scheduling parameters of a thread.

### Syntax

```
pthread_getschedparam(
    thread,
    policy,
    param );
```

Argument	Data Type	Access
thread	opaque pthread_t	read
policy	integer	write
param	struct sched_param	write

### C Binding

```
#include <pthread.h>

int
pthread_getschedparam (
    pthread_t  thread,
    int  *policy,
    struct sched_param  *param);
```

### Arguments

#### thread

Thread whose scheduling policy and parameters are obtained.

#### policy

Receives the value of the scheduling policy for the thread specified in *thread*. Refer to the description of the pthread\_setschedparam function for valid parameters and their values.

#### param

Receives the value of the scheduling parameters for the thread specified in *thread*. Refer to the description of the pthread\_setschedparam function for valid values.

### Description

This routine obtains both the current scheduling policy and associated scheduling parameters of the thread specified by *thread*.

The priority value returned in the *param* structure is the value specified in *attr* at pthread\_create or by the most recent pthread\_setschedparam call affecting the target thread.

This routine differs from pthread\_attr\_getschedpolicy and pthread\_attr\_getschedparam in that those routines get the scheduling policy and parameter attributes that are used to establish the priority and scheduling policy of a new thread when it is created. This routine, however, obtains the scheduling policy and parameters of an existing thread.



## Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[ESRCH]	The value specified by <i>thread</i> does not refer to an existing thread.

## Associated Routines

pthread\_create  
 pthread\_self  
 pthread\_attr\_getschedpolicy  
 pthread\_attr\_getschedparam  
 pthread\_setschedparam



## pthread\_getsequence\_np

---

### pthread\_getsequence\_np

Obtains the *thread* sequence number.

#### Syntax

```
pthread_getsequence_np(  
    thread );
```

Argument	Data Type	Access
thread	opaque pthread_t	read

#### C Binding

```
#include <pthread.h>  
  
unsigned long  
pthread_getsequence_np (  
    pthread_t  *thread);
```

#### Arguments

**thread**

Receives the value for the *thread* sequence number.

#### Description

This routine obtains the *thread* sequence number, which provides a unique identifier for each concurrent thread. Thread sequence numbers are never reused while a thread exists, but may be reused after the thread terminates. The debugger interfaces use this sequence number to identify each thread in commands and in display output.

#### Return Values

No errors are returned. The function `pthread_getsequence_np` returns the sequence number of the specified *thread*. The result is undefined if *thread* is not a valid thread.

#### Associated Routines

`pthread_create`



---

## pthread\_getspecific

Obtains the thread-specific data associated with the specified key.

### Syntax

```
pthread_getspecific(
    key);
```

Argument	Data Type	Access
key	opaque pthread_key_t	read

### C Binding

```
#include <pthread.h>

void
*pthread_getspecific (
    pthread_key_t    key);
```

### Arguments

#### key

Context *key* identifies the thread-specific data to be obtained. This key must be obtained from pthread\_key\_create.

### Description

This routine obtains the thread-specific data associated with the specified *key* for the current thread. This function returns the value currently bound to the specified *key* on behalf of the calling thread.

This routine may be called from a thread-specific data destructor function.

### Return Values

No errors are returned. The function pthread\_getspecific returns the thread-specific data value associated with the given *key*. If no thread-specific data value is associated with *key*, or if *key* is not defined, then a NULL value is returned.



---

## pthread\_join

---

### pthread\_join32, pthread\_join64

---

*The pthread\_join32 and pthread\_join64 forms are only valid in 64-bit pointer environments for OpenVMS Alpha. For information regarding 32- and 64-bit pointers, see Appendix B. Ensure that your compiler provides 64-bit support prior to using pthread\_join64.*

---

Causes the calling thread to wait for the termination of a specified thread.

### Syntax

```
pthread_join(
    thread,
    value_ptr);
```

Argument	Data Type	Access
thread	opaque pthread_t	read
value_ptr	void *	write

### C Binding

```
#include <pthread.h>

int
pthread_join (
    pthread_t  thread,
    void      **value_ptr);
```

### Arguments

#### thread

Thread whose termination is awaited by the caller of this routine.

#### value\_ptr

Return value of the terminating thread (when that thread calls pthread\_exit or returns.)

### Description

This routine suspends execution of the calling thread until the target thread terminates.

A call to a pthread\_join routine returns after the specified thread terminates. The pthread\_join routine is a deferred cancellation point: the target thread will not be detached if the thread blocked in pthread\_join is canceled.



On return from a successful `pthread_join` call with a non-NULL *value\_ptr* argument, the value passed to `pthread_exit` is returned in the location referenced by *value\_ptr*, and the terminating thread is detached. If more than one thread attempts to join with a single thread, the results are unpredictable.

If a thread calls this routine and specifies its own `pthread_t`, a deadlock can result.

The `pthread_join` (or `pthread_detach`) function should eventually be called for every thread that is created with the *detachstate* set to `PTHREAD_CREATE_JOINABLE` so that storage associated with the thread may be reclaimed.

For OpenVMS Alpha systems only, you can call `pthread_join32` or `pthread_join64` instead of `pthread_join`. The `pthread_join32` form returns a 32-bit "void \*" in the address to which *value\_ptr* points. The `pthread_join64` form returns a 64-bit "void \*". You can call either, or you can call `pthread_join`. The `pthread_join` routine is defined to `pthread_join64` if you compile using `/pointer_size=long`. If you do not specify `/pointer_size`, or if you specify `/pointer_size=short`, then `pthread_join` is defined to be `pthread_join32`.

## Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>thread</i> does not refer to a joinable thread.
[ESRCH]	The value specified by <i>thread</i> does not refer to an existing thread ID.
[EDEADLK]	A deadlock was detected or the value of <i>thread</i> specifies the calling thread.

## Associated Routines

`pthread_cancel`  
`pthread_create`  
`pthread_detach`  
`pthread_exit`



## pthread\_key\_create

---

### pthread\_key\_create

Generates a unique thread-specific data key.

#### Syntax

```
pthread_key_create(  
    key,  
    destructor );
```

Argument	Data Type	Access
key	opaque pthread_key_t	write
destructor	procedure	read

#### C Binding

```
#include <pthread.h>  
  
int  
pthread_key_create (  
    pthread_key_t  *key,  
    void  (*destructor)(void *));
```

#### Arguments

##### key

The new thread-specific data key.

##### destructor

Procedure called to destroy a thread-specific data value associated with the created key when the thread terminates. Note, the argument to the destructor for the user-specified routine is the non-null value associated with a key.

#### Description

This routine generates a unique thread-specific data key visible to all threads in the process. The variable *key* provided by this routine is an opaque object used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by `pthread_setspecific` are maintained on a per-thread basis and persist for the life of the calling thread.

Thread-specific data allows client software to associate "static" information with the current thread. For example, where a routine declares a variable "static" in a single threaded program, a multithreaded version of the program might create a thread-specific data key to store the same variable.

This routine generates and returns a new key value. The key reserves a cell within each thread. Each call to this routine creates a new cell that is unique within an application invocation. Keys must be generated from initialization code that is guaranteed to be called only once within each process. (See the `pthread_once` description for more information.)



When a thread terminates, its thread-specific data is automatically destroyed; however, the key remains unless destroyed by a call to `pthread_key_delete`. An optional destructor function may be associated with each key. At thread exit, if a key has a non-NULL destructor pointer, and the thread has a non-NULL value associated with that key, the destructor function is called with the current associated value as its sole argument. The order in which thread-specific data destructors are called at thread termination is undefined.

Before each destructor is called, the thread's value for the corresponding key is set to NULL. If, after the destructors have been called for all non-NULL values with associated destructors, there are still some non-NULL values with associated destructors, then the process is repeated. If there are still non-NULL values for any key with a destructor after four repetitions of this process, DECThreads will terminate the thread. At this point, any key values that represent allocated heap will be lost. Note that this occurs only when a destructor performs some action that creates a new value for some key. Destructor code should attempt to avoid this sort of circularity.

## Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacked the necessary resources to create another thread-specific data key, or the system imposed limit on the total number of keys per process (PTHREAD_KEYS_MAX) has been exceeded.
[ENOMEM]	Insufficient memory exists to create the key.

## Associated Routines

pthread\_getspecific  
pthread\_setspecific  
pthread\_key\_delete  
pthread\_once



---

### pthread\_key\_delete

Deletes a thread-specific data key.

#### Syntax

```
pthread_key_delete(  
    key );
```

Argument	Data Type	Access
key	opaque pthread_key_t	write

#### C Binding

```
#include <pthread.h>  
  
int  
pthread_key_delete (  
    pthread_key_t  key);
```

#### Arguments

**key**  
Context key to be deleted.

#### Description

This routine deletes a thread-specific data *key* previously returned by `pthread_key_create()`. The thread-specific data values associated with *key* need not be NULL at the time this routine is called. The application must free any application storage or perform any cleanup actions for data structures related to the deleted key or associated thread-specific data in any threads. This cleanup can be done before or after this routine call. Do not attempt to use the key after calling this routine since this results in unpredictable behavior.

No destructor functions are invoked by this routine. Any destructor functions that may have been associated with *key*, shall no longer be called upon thread exit. `pthread_key_delete` can be called from within destructor functions.

#### Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The key value is an invalid argument.



## Associated Routines

pthread\_getspecific  
pthread\_key\_create  
pthread\_exit



## pthread\_kill

Delivers a signal to a specified thread.

*This routine is for Digital UNIX systems only.*

### Syntax

```
pthread_kill(
    thread,
    sig );
```

Argument	Data Type	Access
thread	opaque pthread_t	read
sig	integer	read

### C Binding

```
#include <pthread.h>
```

```
int
```

```
pthread_kill (
    pthread_t  thread,
    int  sig);
```

### Arguments

#### thread

Thread to receive a signal request.

#### sig

A signal request. If *sig* is zero, error checking is performed, but no signal is sent.

### Description

This routine sends a signal to the specified thread. Any signal defined to stop, continue, or terminate will stop or terminate the process, even though it may be handled by the thread. For example, SIGTERM terminates all threads in the process, even though it may be handled by the thread to which it is sent. The name of the "kill" routine is sometimes misleading because many signals do not terminate a thread.

The various signals are as follows:

SIGHUP	SIGPIPE	SIGTTIN
SIGINT	SIGALRM	SIGTTOU
SIGQUIT	SIGTERM	SIGIO
SIGTRAP	SIGUSR1	SIGXCPU
SIGABRT	SIGSYS	SIGXFSZ
SIGEMT	SIGURG	SIGVTALRM
SIGFPE	SIGSTOP	SIGPROF



SIGKILL  
SIGBUS  
SIGSEGV

SIGTSTP  
SIGCONT  
SIGCHLD

SIGINFO  
SIGUSR1  
SIGUSR2

If this routine does not execute successfully, no signal is sent.

## Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value of <i>sig</i> is invalid or an unsupported signal value.
[ESRCH]	The value of <i>thread</i> does not specify an existing thread.



---

### pthread\_lock\_global\_np

Locks the global mutex if the global mutex is unlocked. If the global mutex is locked by another thread, causes the thread to wait for the global mutex to become available.

#### Syntax

```
pthread_lock_global_np( );
```

#### C Binding

```
#include <pthread.h>

int
pthread_lock_global_np (void);
```

#### Arguments

None

#### Description

This routine locks the global mutex. If the global mutex is currently held by another thread when a thread calls this routine, the thread waits for the global mutex to become available.

The thread that has locked the global mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the global mutex in the locked state and with the current thread as the global mutex's current owner.

Use the global mutex when calling a library package that is not designed to run in a multithreaded environment. (Unless the documentation for a library function specifically states that it is thread safe, assume that it is not compatible; in other words, assume it is nonreentrant.)

The global mutex is one lock. Any code that calls any function that is not known to be reentrant uses the same lock. This prevents problems resulting from dependencies among threads calling library functions and those functions calling other functions, and so on.

The global mutex is a recursive mutex. A thread that has locked the global mutex can relock it without deadlocking. (The locking thread must call `pthread_unlock_global_np` as many times as it called this routine to allow another thread to lock the global mutex.)



## Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.

## Associated Routines

`pthread_unlock_global_np`



## pthread\_mutex\_destroy

---

## pthread\_mutex\_destroy

Destroys a mutex.

### Syntax

```
pthread_mutex_destroy(  
    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	write

### C Binding

```
#include <pthread.h>  
  
int  
pthread_mutex_destroy (  
    pthread_mutex_t  *mutex);
```

### Arguments

**mutex**  
The mutex to be destroyed.

### Description

This routine destroys a mutex by uninitializing it, and should be called when a mutex object is no longer referenced. This routine may reclaim internal storage used by the mutex object.

It is safe to destroy an initialized mutex that is unlocked. However, it is illegal to destroy a locked mutex.

The results of this routine are unpredictable if the mutex object specified in the *mutex* argument does not currently exist, or is not initialized.



## Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EBUSY]	An attempt is made to destroy the object referenced by <i>mutex</i> while it is locked or referenced.
[EINVAL]	The value specified by <i>mutex</i> is invalid.

## Associated Routines

pthread\_mutex\_init  
pthread\_mutex\_lock  
pthread\_mutex\_trylock  
pthread\_mutex\_unlock



## pthread\_mutex\_init

Initializes a mutex with attributes specified by the *attr* argument.

### Syntax

```
pthread_mutex_init(
    mutex,
    attr );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	write
attr	opaque pthread_mutexattr_t	read

### C Binding

```
#include <pthread.h>

int
pthread_mutex_init (
    pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);
```

### Arguments

**mutex**  
Mutex created.

**attr**  
Mutex attributes object to be used in initializing the characteristics of the created *mutex*.

### Description

This routine initializes a mutex with the attributes specified by the *attr* argument. A mutex is a synchronization object that allows multiple threads to serialize their access to shared data.

The mutex is initialized and set to the unlocked state. If *attr* is set to NULL, the default mutex attributes are used. The `pthread_mutexattr_settype_np` routine can be used to specify the type of mutex that is created (normal, recursive, or errorcheck).

See Chapter 2 for more information about mutex usage.

A mutex is a resource of the process, not part of any particular thread. A mutex is neither destroyed nor unlocked automatically when any thread exits. Because mutexes are shared, they may be allocated in heap or static memory but not on a stack.

The `PTHREAD_MUTEX_INITIALIZER` macro can be used to statically initialize a mutex without calling this routine. Statically initialized mutexes need not be destroyed using `pthread_mutex_destroy`. Use this macro as follows:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER
```



Only normal mutexes can be statically initialized.

## Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error, the mutex is not initialized, and the contents of *mutex* are undefined. Possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacks the necessary resources to initialize a mutex.
[ENOMEM]	Insufficient memory exists to initialize the mutex.
[EBUSY]	The implementation has detected an attempt to reinitialize the <i>mutex</i> (a previously initialized, but not yet destroyed mutex).
[EINVAL]	The value specified by <i>mutex</i> is invalid.
[EPERM]	The caller does not have the privileges to perform this operation.

## Associated Routines

pthread\_mutexattr\_init  
 pthread\_mutexattr\_gettype\_np  
 pthread\_mutexattr\_settype\_np  
 pthread\_mutex\_lock  
 pthread\_mutex\_trylock  
 pthread\_mutex\_unlock



---

## pthread\_mutex\_lock

Locks an unlocked mutex. If the mutex is already locked, the calling thread blocks until the mutex becomes available.

### Syntax

```
pthread_mutex_lock(  
    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

### C Binding

```
#include <pthread.h>  
  
int  
pthread_mutex_lock (  
    pthread_mutex_t *mutex);
```

### Arguments

**mutex**  
Mutex to be locked.

### Description

This routine locks a mutex with varying behavior as follows:

- If a recursive mutex was specified, the current owner of a mutex can relock the same mutex without blocking. The lock count is incremented for each recursive lock within the thread.
- If an error-check mutex was specified and the current owner tries to lock the mutex a second time, the EDEADLK error is reported. If the mutex is locked by another thread, the calling thread waits for the mutex to become available.
- If a normal mutex is specified, a deadlock can result if the current owner of a mutex calls this routine in an attempt to lock the mutex a second time. (The deadlock is not detected or reported.)

Use the pthread\_mutexattr\_settype\_np routine to set the type of the mutex to normal, recursive, or error-check. For more information about mutexes, see Chapter 2.

The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the mutex in the locked state and with the current thread as the mutex's current owner.

A recursive or error-check mutex records the identity of the thread that locks it, allowing debuggers to display this information. In most cases, normal mutexes do not record the thread identity.



## Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>mutex</i> is invalid, or The <i>mutex</i> was created with the protocol attribute set to PTHREAD_PRIO_PROTECT and the calling thread's priority set higher than the mutex's current priority ceiling.
[EDEADLK]	A deadlock condition is detected.

## Associated Routines

pthread\_mutexattr\_settype\_np  
pthread\_mutex\_destroy  
pthread\_mutex\_init  
pthread\_mutex\_trylock  
pthread\_mutex\_unlock



### pthread\_mutex\_trylock

Tries to lock a mutex. If the mutex is already locked, the calling thread does not wait for the mutex to become available.

#### Syntax

```
pthread_mutex_trylock(  
    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

#### C Binding

```
#include <pthread.h>  
  
int  
pthread_mutex_trylock (  
    pthread_mutex_t *mutex);
```

#### Arguments

**mutex**  
Mutex locked.

#### Description

This routine tries to lock a mutex. When a thread calls this routine, an attempt is made to immediately lock the mutex. If the mutex is successfully locked, 0 is returned and the current thread is then the mutex's current owner. If the specified mutex is locked when a thread calls this routine, the calling thread does not wait for the mutex to become available.

The behavior of this routine is as follows:

- If a recursive mutex is owned by the current thread, a zero is returned and the mutex lock count is incremented. (To unlock a recursive mutex, each call to pthread\_mutex\_trylock must be matched by a call to pthread\_mutex\_unlock.)
- If a normal or error-check mutex is locked by any thread (including the current thread) when this routine is called, EBUSY is returned and the thread does not wait to acquire the lock.
- If a normal or error-check mutex is not owned, a zero is returned and the mutex becomes locked.

The pthread\_mutexattr\_settype\_np routine is used to set the mutex *type* attribute (normal, recursive, or errorcheck). For information about mutex types and their usage, see Chapter 2.



## Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EBUSY]	The mutex is already locked; therefore, it was not acquired.
[EINVAL]	The value specified by <i>mutex</i> is invalid, or The <i>mutex</i> was created with the protocol attribute set to PTHREAD_PRIO_PROTECT and the calling thread's priority set higher than the mutex's current priority ceiling.

## Associated Routines

pthread\_mutexattr\_settype\_np  
pthread\_mutex\_destroy  
pthread\_mutex\_init  
pthread\_mutex\_lock  
pthread\_mutex\_unlock



## pthread\_mutex\_unlock

Unlocks a mutex.

### Syntax

```
pthread_mutex_unlock(  
    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

### C Binding

```
#include <pthread.h>  
  
int  
pthread_mutex_unlock (  
    pthread_mutex_t *mutex);
```

### Arguments

**mutex**  
Mutex unlocked.

### Description

This routine unlocks a mutex.

The following describes the varied behavior of this routine:

- When an owner unlocks a recursive mutex that it owns, the lock count is decremented. The mutex remains locked and owned until the count reaches 0. When the lock count reaches 0, or for any other type of mutex, the mutex becomes unlocked with no current owner.
- If a normal or error-check mutex is owned by the current thread, it is unlocked.
- If an error-check mutex is not locked or is locked by another thread, EPERM is returned. A normal mutex may also return EPERM, but this is not guaranteed.

If one or more threads are waiting to lock the specified mutex, and the mutex becomes unlocked, this routine causes one thread to unblock and try to acquire the mutex. The scheduling policy is used to determine which thread to unblock. For the SCHED\_FIFO and SCHED\_RR policies, a blocked thread is chosen in priority order, using FIFO within priorities. Note that the mutex may not be acquired by the awakened thread if any other running thread attempts to lock the mutex first.

On Digital Unix, if a signal is delivered to a thread waiting for a mutex, upon return from the signal handler, the thread resumes waiting for the mutex as if it was not interrupted.



## Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified for <i>mutex</i> is invalid.
[EPERM]	The calling thread does not own the mutex.

## Associated Routines

pthread\_mutexattr\_settype\_np  
 pthread\_mutex\_destroy  
 pthread\_mutex\_init  
 pthread\_mutex\_lock  
 pthread\_mutex\_trylock  
 pthread\_unlock\_global\_np



## pthread\_mutexattr\_destroy

---

## pthread\_mutexattr\_destroy

Deletes a mutex attributes object.

### Syntax

```
pthread_mutexattr_destroy(  
    attr);
```

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	write

### C Binding

```
#include <pthread.h>  
  
int  
pthread_mutexattr_destroy (  
    pthread_mutexattr_t *attr);
```

### Arguments

**attr**  
Attributes object deleted.

### Description

This routine destroys a mutex attributes object (the object becomes uninitialized). Call this routine when a mutex attributes object is no longer needed.

This routine gives permission to reclaim storage for the mutex attributes object. Mutexes that were created using this attributes object are not affected by the deletion of the mutex attributes object.

The results of calling this routine are unpredictable if the attributes object specified in the *attr* argument does not exist.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The object value specified by <i>attr</i> is invalid.

### Associated Routines

pthread\_mutexattr\_init



---

## pthread\_mutexattr\_gettype\_np

Obtains the mutex type attribute used when a mutex is created.

### Syntax

```
pthread_mutexattr_gettype_np(  
    attr,  
    type );
```

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	read
type	integer	write

### C Binding

```
#include <pthread.h>  
  
int  
pthread_mutexattr_gettype_np (  
    const pthread_mutexattr_t  *attr,  
    int  *type);
```

### Arguments

**attr**

Mutex attributes object whose mutex type is obtained.

**type**

Gets the value of the mutex type attribute. The *type* argument specifies the type of mutex that is created. Valid values are:

- PTHREAD\_MUTEX\_NORMAL\_NP (default)
- PTHREAD\_MUTEX\_RECURSIVE\_NP
- PTHREAD\_MUTEX\_ERRORCHECK\_NP

### Description

This routine obtains the mutex *type* attribute that is used when a mutex is created. See the pthread\_mutexattr\_settype\_np description for information about mutex type attributes.

### Return Values

On successful completion, this routine returns the mutex type in *type*.

If an error condition occurs, this routine returns an integer value indicating the type of the error. Possible return values are as follows:

Return	Description
0	Successful completion.



## pthread\_mutexattr\_gettype\_np

Return	Description
[EINVAL]	The value specified by <i>attr</i> is invalid.

### Associated Routines

pthread\_mutexattr\_init  
pthread\_mutexattr\_settype\_np  
pthread\_mutex\_init



## pthread\_mutexattr\_init

Initializes a mutex attributes object that is used to specify the attributes of mutexes when they are created.

### Syntax

```
pthread_mutexattr_init(
    attr );
```

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	write

### C Binding

```
#include <pthread.h>

int
pthread_mutexattr_init (
    pthread_mutexattr_t  *attr);
```

### Arguments

**attr**  
Mutex attributes object created.

### Description

This routine initializes a mutex attributes object (*attr*) used to specify the attributes of mutexes when they are created. The mutex attributes object is initialized with the default value for all of the attributes defined by DECthreads.

When a mutex attributes object is used to initialize a mutex, the values of the individual attributes determine the characteristics of the new object. Attributes objects act like additional arguments to object creation. Changing individual attributes or destroying the attributes object does not affect any objects that were previously created using the attributes object.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[ENOMEM]	Insufficient memory while attempting to create the mutex attributes object.



Associated Routines

pthread\_create  
pthread\_mutexattr\_gettype\_np  
pthread\_mutexattr\_settype\_np  
pthread\_mutex\_init

Argument	Data Type	Default
mutexattr	pthread_mutexattr_t	NULL

C Binding

Arguments

Description

Return Values

Return	Description
0	Successful completion.
EINVAL	Invalid argument.



## pthread\_mutexattr\_settype\_np

Specifies the mutex type attribute that is used when a mutex is created.

### Syntax

```
pthread_mutexattr_settype_np(
    attr,
    type );
```

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	write
type	integer	read

### C Binding

```
#include <pthread.h>

int
pthread_mutexattr_settype_np (
    pthread_mutexattr_t *attr,
    int type);
```

### Arguments

**attr**  
Mutex attributes object modified.

**type**  
New value for the mutex type attribute. The *type* argument specifies the type of mutex that is created. Valid values are:

- PTHREAD\_MUTEX\_NORMAL\_NP (default)
- PTHREAD\_MUTEX\_RECURSIVE\_NP
- PTHREAD\_MUTEX\_ERRORCHECK\_NP

### Description

This routine sets the mutex type attribute that is used to determine which type of mutex is created on a call to pthread\_mutex\_init. See Section 2.2.4.1 for information on the types of mutexes.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>attr</i> or <i>type</i> is invalid.



## pthread\_mutexattr\_settype\_np

Return	Description
[ESRCH]	The value specified by <i>attr</i> does not refer to an existing mutex attributes object.

### Associated Routines

pthread\_mutexattr\_init  
pthread\_mutexattr\_gettype\_np  
pthread\_mutex\_init

Return	Type	Description
0	int	Success
[ESRCH]	int	The value specified by <i>attr</i> does not refer to an existing mutex attributes object.



## pthread\_once

Calls an initialization routine that can be executed by a single thread, once.

### Syntax

```
pthread_once(
    once_control,
    init_routine );
```

Argument	Data Type	Access
once_control	opaque pthread_once_t	modify
init_routine	procedure	read

### C Binding

```
#include <pthread.h>

int
pthread_once (
    pthread_once_t  *once_control,
    void  (*init_routine) (void));
```

### Arguments

#### once\_control

Address of a record that defines the one-time initialization code. Each one-time initialization routine must have its own unique pthread\_once\_t record.

#### init\_routine

Address of a procedure that performs the initialization. This routine is called only once, regardless of the number of times it and its associated *once\_control* are passed to pthread\_once.

### Description

The first call to this routine by any thread in a process with a given *once\_control* will call the *init\_routine()* with no arguments. Then subsequent calls to *pthread\_once()* with the same *once\_control* will not call the *init\_routine()*. On return from *pthread\_once()*, it is guaranteed that the initialization routine has completed.

For example, a mutex or a per-thread context key must be created exactly once. Calling *pthread\_once* ensures that the initialization is serialized across multiple threads. Other threads that reach the same point in the code would be delayed until the first thread is finished.

The *pthread\_once\_t* variable must be statically initialized using the *PTHREAD\_ONCE\_INIT* macro or by zeroing out the entire structure.



### Note

If you specify an *init\_routine* that directly or indirectly results in a recursive call to `pthread_once` specifying the same *init\_block* argument, the recursive call may result in a deadlock.

The `PTHREAD_ONCE_INIT` macro, defined in `<pthread.h>` header file, must be used to initialize a once control record. A *once\_control* record must be declared as follows:

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

Note that it is often easier to simply lock a statically initialized mutex, check a control flag, and perform necessary initialization (in-line) rather than using `pthread_once`. For example, code an init routine beginning with the following basic logic:

```
init()
{
    static pthread_mutex_t mutex = PTHREAD_MUTEX_INIT;
    static int flag = FALSE;

    pthread_mutex_lock(&mutex);
    if(!flag)
    {
        flag = TRUE;
        /* initialize code */
    }
    pthread_mutex_unlock(&mutex);
}
```

### Return Values

If an error occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	Invalid argument.



---

## pthread\_self

Obtains the identifier of the current thread.

### Syntax

```
pthread_self( );
```

### C Binding

```
#include <pthread.h>
pthread_t
pthread_self(void);
```

### Arguments

None

### Description

This routine allows a thread to obtain its own thread identifier.

This value becomes meaningless when the thread is deleted.

### Return Values

Returns the identifier of the calling thread.

### Associated Routines

pthread\_cancel  
pthread\_create  
pthread\_detach  
pthread\_exit  
pthread\_join  
pthread\_kill  
pthread\_sigmask



## pthread\_setcancelstate

---

## pthread\_setcancelstate

Sets the current thread's cancelability state.

### Syntax

```
pthread_setcancelstate(  
    state,  
    oldstate );
```

Argument	Data Type	Access
state	integer	read
oldstate	integer	write

### C Binding

```
#include <pthread.h>
```

```
int
```

```
pthread_setcancelstate (  
    int state,  
    int *oldstate );
```

### Arguments

#### state

State of general cancelability to set for the calling thread. The following are valid cancel state values:

- PTHREAD\_CANCEL\_ENABLE
- PTHREAD\_CANCEL\_DISABLE

#### oldstate

Previous cancelability state.

### Description

This routine sets the current thread's cancelability state and returns the previous cancelability state in *oldstate*.

When cancelability state is set to PTHREAD\_CANCEL\_DISABLE, a cancel cannot be delivered to the thread, even if a cancelable routine is called or asynchronous cancelability type is enabled.

When a thread is created, the default general cancelability state is PTHREAD\_CANCEL\_ENABLE.

#### Possible Problems When Disabling Cancelability

The most important use of cancel calls is to ensure that indefinite wait operations are terminated. For example, a thread waiting on some network connection, which may take days to respond (or may never respond), should be made cancelable.



When cancelability is disabled, no routine is cancelable. As a result, the user is unable to cancel the operation. When disabling cancelability, be sure that no long waits can occur or that it is necessary for other reasons to defer cancels around that particular region of code.

## Return Values

On successful completion, this routine returns the previous state of general cancelability in *oldstate*.

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The specified state is not PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.

## Associated Routines

pthread\_cancel  
pthread\_setcanceltype  
pthread\_testcancel



## pthread\_setcanceltype

---

## pthread\_setcanceltype

Sets the current thread's cancelability type.

### Syntax

```
pthread_setcanceltype(  
    type,  
    oldtype );
```

Argument	Data Type	Access
type	integer	read
oldtype	integer	write

### C Binding

```
#include <pthread.h>  
  
int  
pthread_setcanceltype (  
    int  type,  
    int  *oldtype);
```

### Arguments

#### type

The cancelability type to set for the calling thread. The following are valid values:

- PTHREAD\_CANCEL\_DEFERRED
- PTHREAD\_CANCEL\_ASYNCHRONOUS

#### oldtype

Returns the previous cancelability type.

### Description

This routine sets the cancelability type and returns the previous type at location *oldtype*.

When the cancelability state is set to PTHREAD\_CANCEL\_DISABLE, (see pthread\_setcancelstate()), a cancel cannot be delivered to the thread, even if a cancelable routine is called or asynchronous cancelability type is enabled.

When the cancelability state is set to PTHREAD\_CANCEL\_ENABLE, cancelability depends on the thread's cancelability type. When the thread's cancelability state is PTHREAD\_CANCEL\_ENABLE and the thread's cancelability type is set to PTHREAD\_CANCEL\_DEFERRED, the thread can only receive a cancel at specific cancellation points (including condition waits, thread joins, and calls to pthread\_testcancel.) If the thread's cancelability state is PTHREAD\_CANCEL\_ENABLE and its cancelability type is PTHREAD\_CANCEL\_ASYNCHRONOUS, the thread can be canceled at any point in its execution.



When a thread is created, the default cancelability type is `PTHREAD_CANCEL_DEFERRED`.

---

### Warning

---

If the asynchronous cancelability type is set, do not call any routine unless it is explicitly documented as safe to be called with the asynchronous cancelability type. Note that none of the general run-time libraries and none of the DECThreads libraries are safe except for `pthread_setcanceltype`, `pthread_setcancelstate`, and `cma_alert_restore`.

The asynchronous cancelability type should only be used when you have a compute bound section of code that carries no state and makes no routine calls.

---

## Return Values

On successful completion, this routine returns the previous cancelability type in *oldtype*.

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The specified type is not <code>PTHREAD_CANCEL_DEFERRED</code> or <code>PTHREAD_CANCEL_AYNCHRONOUS</code> .

## Associated Routines

pthread\_cancel  
pthread\_setcancelstate  
pthread\_testcancel



## pthread\_setschedparam

### pthread\_setschedparam

Changes the current scheduling policy and scheduling parameters of a thread.

#### Syntax

```
pthread_setschedparam(  
    thread,  
    policy,  
    param );
```

Argument	Data Type	Access
thread	opaque pthread_t	read
policy	integer	read
param	struct sched_param	read

#### C Binding

```
#include <pthread.h>  
  
int  
pthread_setschedparam (  
    pthread_t  thread,  
    int  policy,  
    const struct sched_param  *param);
```

#### Arguments

##### thread

Thread whose scheduling policy and parameters are to be changed.

##### policy

New scheduling policy value for the thread specified in *thread*. The following are valid values:

```
SCHED_BG_NP  
SCHED_FG_NP  
SCHED_FIFO  
SCHED_OTHER  
SCHED_RR
```

See Section 2.2.3.2 for a description of the scheduling policies.

##### param

New values of the scheduling parameters associated with the scheduling policy for the thread specified in *thread*. Valid values for the sched\_priority field of struct sched\_param depend on the chosen policy and fall within one of the following ranges.



Low	High
PRI_FIFO_MIN	PRI_FIFO_MAX
PRI_RR_MIN	PRI_RR_MAX
PRI_OTHER_MIN	PRI_OTHER_MAX
PRI_FG_MIN_NP	PRI_FG_MAX_NP
PRI_BG_MIN_NP	PRI_BG_MAX_NP

Calculate the priority using the appropriate symbols such as `PRI_FIFO_MIN` or `PRI_FIFO_MAX`. Avoid using numeric constants as priorities. (Section 2.7 describes how to specify priorities between the minimum and maximum values.)

## Description

This routine changes both the current scheduling policy and associated scheduling parameters of the thread specified by *thread* to the policy and associated parameters provided in *policy* and *param*, respectively.

All currently implemented DECthreads scheduling policies have one scheduling parameter called `sched_priority`. You must specify an appropriate `sched_priority` parameter for the policy you choose.

Changing the scheduling policy or priority, or both, of a thread can cause it to start executing or to be preempted by another thread. A thread changes its own scheduling policy and priority by using the handle returned by `pthread_self`.

This routine differs from `pthread_attr_setschedpolicy` and `pthread_attr_setschedparam` in that those routines set the scheduling policy and parameter attributes that are used to establish the priority and scheduling policy of a new thread when it is created. This routine, however, changes the scheduling policy and parameters of an existing thread.

## Return Values

If an error condition occurs, no scheduling policy or parameters are changed for the target thread and this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>policy</i> or <i>param</i> is invalid.
[ENOTSUP]	An attempt is made to set the scheduling policy or a parameter to an unsupported value.
[EPERM]	The caller does not have the appropriate privileges to set the scheduling policy or parameters of the specified thread.
[ESRCH]	The value specified by <i>thread</i> does not refer to an existing thread.



## Associated Routines

pthread\_attr\_setschedparam  
pthread\_attr\_setschedpolicy  
pthread\_create  
pthread\_self



## pthread\_setspecific

Sets the thread-specific data value associated with the specified key for the current thread.

### Syntax

```
pthread_setspecific(
    key,
    value );
```

Argument	Data Type	Access
key	opaque pthread_key_t	read
value	void *	read

### C Binding

```
#include <pthread.h>

int
pthread_setspecific (
    pthread_key_t  key,
    const void  *value);
```

### Arguments

#### key

Thread-specific key that identifies the thread-specific data to receive *value*. This key value must be obtained from pthread\_key\_create.

#### value

New thread-specific data value to associate with the specified key for the current thread.

### Description

This routine sets the thread-specific data value associated with the specified key for the current thread. If a value is defined for the key in this thread (the current value is not NULL), the new value is substituted for it. The key is obtained by a previous call to pthread\_key\_create.

Different threads can bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that are reserved for use by the calling thread.

This routine may be called from a thread-specific data destructor function. However, calling this routine from a destructor may result in lost storage or infinite loops.

Note that although the type for *value* (void \*) implies an address, the type is being used as a "universal scalar type". DECthreads does nothing with *value* other than store it for later retrieval.



## pthread\_setspecific

### Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The specified <i>key</i> is invalid.
[ENOMEM]	Insufficient memory exists to associate the value with the key.

### Associated Routines

pthread\_getspecific  
pthread\_key\_create  
pthread\_key\_delete



## pthread\_sigmask

Examine or change the current thread's signal mask.

*This routine is for Digital UNIX systems only.*

### Syntax

```
pthread_sigmask(
    how,
    set,
    oset);
```

Argument	Data Type	Access
how	integer	read
set	sigset_t	read
oset	sigset_t	write

### C Binding

```
#include <pthread.h>

int
pthread_sigmask (
    int how,
    const sigset_t *set,
    sigset_t *oset);
```

### Arguments

#### how

Indicates the manner in which the set of masked signals is changed. The optional values are as follows:

SIG_BLOCK	The resulting set is the union of the current set and the signal set pointed to by the <i>set</i> argument.
SIG_UNBLOCK	The resulting set is the intersection of the current set and the complement of the signal set pointed to by the <i>set</i> argument.
SIG_SETMASK	The resulting set is the signal set pointed to by the <i>set</i> argument.

#### set

Specifies the signal set by pointing to a set of signals used to change the blocked set. If this *set* value is NULL, the *how* argument is ignored and the process signal mask is unchanged.

#### oset

Receives the value of the current signal mask (unless this value is NULL).



### Description

This routine examines or changes the calling thread's signal mask. Typically, you use the SIG\_BLOCK option for the *how* value to block signals during a critical section of code, and then use the SIG\_SETMASK option of this routine to restore the mask to the previous value returned by the previous call to the pthread\_sigmask() function.

If there are any unblocked signals pending after a call to this routine, at least one of those signals will be delivered before this routine returns.

This routine does not allow the SIGKILL or SIGSTOP signals to be blocked. If a program attempts to block one of these signals, pthread\_sigmask function gives no indication of the error.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified for <i>how</i> is invalid.



---

## pthread\_testcancel

Requests delivery of any pending cancel to the current thread.

### Syntax

```
pthread_testcancel( );
```

### C Binding

```
#include <pthread.h>

void
pthread_testcancel (void);
```

### Arguments

None

### Description

This routine requests delivery of a pending cancel to the current thread. The cancel is delivered only if a cancel is pending for the current thread and the cancelability state is enabled. (A thread disables delivery of cancels to itself by calling `pthread_setcancelstate`.)

This routine, when called within very long loops, ensures that a pending cancel is noticed within a reasonable amount of time.

### Return Values

None



## pthread\_unlock\_global\_np

---

### pthread\_unlock\_global\_np

Unlocks a global mutex.

#### Syntax

```
pthread_unlock_global_np( );
```

#### C Binding

```
#include <pthread.h>

int
pthread_unlock_global_np (void);
```

#### Arguments

None

#### Description

This routine unlocks the global mutex. Since the global mutex is recursive, the unlock will occur when each call to `pthread_lock_global_np` has been matched by a call to this routine. For example, if you called `pthread_lock_global_np` three times, `pthread_unlock_global_np` unlocks the global mutex when you call it the third time. If no threads are waiting for the global mutex, it becomes unlocked with no current owner. If one or more threads are waiting to lock the global mutex, this routine causes one thread to unblock and try to acquire the mutex. The scheduling policy is used to determine which thread is awakened. For the policies `SCHED_FIFO` and `SCHED_RR`, a blocked thread is chosen in priority order, using FIFO within priorities.

#### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EPERM]	The mutex is unlocked or owned by another thread.

#### Associated Routines

`pthread_lock_global_np`



---

## sched\_yield

Notifies the scheduler that the current thread is willing to release its processor to other threads of the same or higher priority.

### Syntax

```
sched_yield( );
```

### C Binding

```
#include <pthread.h>

void
sched_yield(void);
```

### Arguments

None

### Description

This routine forces the running thread to relinquish the processor until it again becomes the head of its thread list. This routine notifies the thread scheduler that the current thread is willing to release its processor to other threads of equivalent or greater scheduling precedence. (A thread generally will release its processor to a thread of a greater scheduling precedence without calling this routine.) If no other threads of equivalent or greater scheduling precedence are ready to execute, the thread continues.

This routine can allow knowledge of the details of an application to be used to improve its performance. If a thread does not call `sched_yield`, other threads may be given the opportunity to run at arbitrary points (possibly even when the interrupted thread holds a required resource). By making strategic calls to `sched_yield`, other threads can be given the opportunity to run when the resources are free. This can sometimes improve performance by reducing contention for the resource.

As a general guideline, consider calling this routine after a thread has released a resource (such as a mutex) that is heavily contended for by other threads. This can be especially important if the program is running on a uniprocessor machine, or if the thread acquires and releases the resource inside a tight loop.

Use this routine carefully and sparingly, because misuse can cause unnecessary context switching which will increase overhead and degrade performance. For example, it is counter-productive for a thread to yield while it holds a resource that the threads to which it is yielding will need. Likewise, it is pointless to yield unless there is likely to be another thread that is ready to run.

### Return Values

None



## Associated Routines

pthread\_attr\_setschedparam  
pthread\_setschedparam



# Part III

---

## Digital Proprietary Interfaces: tis Routines Reference

Part III provides detailed descriptions of the thread-independent services (**tis**) routines that comprise a Digital proprietary interface of DECthreads.

The **tis** routines are designed to provide efficient tools for thread safety in libraries that do not create threads. The **tis** interface provides functions that are identical to the most common pthread functions. In a program using threads, the **tis** functions provide full thread synchronization and memory coherence. But, when run in a program that does not use threads, the same **tis** calls provide low-overhead "stub" implementations of pthread features.

The **tis** objects created using this interface are the same as DECthreads core (POSIX 1003.1c) objects.

In a nonthreaded environment, condition variables should never be used to block operations. (For example, the single-threaded implementation of `tis_cond_wait` cannot block. If it did, no other thread would be running to wake the blocking thread.)

When threads are present, the guidelines for using the pthread routines apply to the use of the corresponding **tis** routine.

Note that *errno* is not used by the **tis** routines. Like the pthread routines, the **tis** routines return integer values indicating the type of error.







---

## tis\_cond\_broadcast

Wakes all threads that are waiting on a condition variable.

### Syntax

```
tis_cond_broadcast(
    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify

### C Binding

```
#include <tis.h>

int
tis_cond_broadcast (
    pthread_cond_t  *cond);
```

### Arguments

#### **cond**

Address of the condition variable (passed by reference) on which to broadcast.

### Description

When threads are not present, `tis_cond_broadcast` performs no actions.

When threads are present, `tis_cond_broadcast` unblocks all threads waiting on the specified condition variable *cond*. For further information about actions when threads are present, refer to the `pthread_cond_broadcast` description.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>cond</i> is invalid.

### Associated Routines

```
tis_cond_destroy
tis_cond_init
tis_cond_signal
tis_cond_wait
```



---

## tis\_cond\_destroy

Destroys a condition variable.

### Syntax

```
tis_cond_destroy(
    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	write

### C Binding

```
#include <tis.h>

int
tis_cond_destroy (
    pthread_cond_t *cond);
```

### Arguments

#### **cond**

Address of the condition variable (passed by reference) to be destroyed.

### Description

This routine destroys the condition variable specified by *cond*. This effectively uninitializes the condition variable. Call this routine when a condition variable will no longer be referenced.

The results of this routine are unpredictable if the condition variable specified in *cond* does not exist or is not initialized.

For more information about actions when threads are present, refer to the [pthread\\_cond\\_destroy](#) description.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>cond</i> is invalid.
[EBUSY]	The object being referenced by <i>cond</i> is being referenced by another thread that is currently executing a <i>tis_cond_wait</i> on the condition variable specified in <i>cond</i> . (This error can only occur when threads are present.)



## Associated Routines

tis\_cond\_broadcast  
 tis\_cond\_init  
 tis\_cond\_signal  
 tis\_cond\_wait



## tis\_cond\_init

Initializes a condition variable.

### Syntax

```
tis_cond_init(
    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	write

### C Binding

```
#include <tis.h>

int
tis_cond_init (
    pthread_cond_t  *cond);
```

### Arguments

#### **cond**

Address of the condition variable (passed by reference) to be initialized.

### Description

This routine initializes a condition variable (*cond*) with the default condition variable attributes.

A condition variable is a synchronization object used in conjunction with a mutex. A mutex controls access to shared data. When threads are present, a condition variable allows threads to wait for data to enter a defined state.

For further information about actions when threads are present, refer to the [pthread\\_cond\\_init](#) description.

The macro `PTHREAD_COND_INITIALIZER` can be used to initialize statically allocated condition variables to the default condition variable attributes. Use this macro as follows:

```
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
```

When statically initialized, a condition variable should not also be initialized using `tis_cond_init`. Also, a statically initialized condition variable need not be destroyed using `tis_cond_destroy`.

### Return Values

If there is an error condition, the following occurs:

- The routine returns an integer value indicating the type of error.
- The condition variable is not initialized.
- The contents of `<variable>(cond)` are undefined.



The possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacks the necessary resources to initialize another condition variable, or The system-imposed limit on the total number of condition variables under execution by a single user is exceeded.
[EBUSY]	The implementation has detected an attempt to reinitialize the object referenced by <i>cond</i> , a previously initialized, but not yet destroyed condition variable.
[EINVAL]	The value specified by <i>attr</i> is invalid.
[ENOMEM]	Insufficient memory exists to initialize the condition variable.

### Associated Routines

tis\_cond\_broadcast  
 tis\_cond\_destroy  
 tis\_cond\_signal  
 tis\_cond\_wait



## tis\_cond\_signal

---

## tis\_cond\_signal

Wakes at least one thread that is waiting on a condition variable.

### Syntax

```
tis_cond_signal(  
    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify

### C Binding

```
#include <tis.h>  
  
int  
tis_cond_signal (  
    pthread_cond_t  *cond);
```

### Arguments

**cond**

Address of the condition variable (passed by reference) on which to signal.

### Description

When threads are present, this routine unblocks at least one thread waiting on the specified condition variable *cond*.

For more information about actions when threads are present, refer to the pthread\_cond\_signal description.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>cond</i> is invalid.

### Associated Routines

```
tis_cond_broadcast  
tis_cond_destroy  
tis_cond_init  
tis_cond_wait
```



---

## tis\_cond\_wait

Causes a thread to wait for a condition variable to be signaled or broadcast.

### Syntax

```
tis_cond_wait(
    cond,
    mutex );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	modify
mutex	opaque pthread_mutex_t	modify

### C Binding

```
#include <tis.h>

int
tis_cond_wait (
    pthread_cond_t *cond,
    pthread_mutex_t *mutex);
```

### Arguments

#### cond

Address of the condition variable (passed by reference) on which to wait.

#### mutex

Address of the mutex (passed by reference) which is associated with the condition variable specified in *cond*.

### Description

When threads are present, this routine causes a thread to wait for a condition variable to be signaled or broadcasted.

Calling this routine in a nonthreaded environment is a coding error. As no thread can execute in parallel to issue a `tis_cond_signal` or `tis_cond_broadcast`, using this routine in a nonthreaded environment will force the program to exit.

For further information about actions when threads are present, refer to the `pthread_cond_wait` description.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.



## tis\_cond\_wait

Return	Description
[EINVAL]	The value specified by <i>cond</i> or <i>mutex</i> is invalid, or: Different mutexes are supplied for concurrent <i>tis_cond_wait</i> operations on the same condition variable, or The mutex was not owned by the current thread at the time of the call.

## Associated Routines

tis\_cond\_broadcast  
tis\_cond\_destroy  
tis\_cond\_init  
tis\_cond\_signal



---

## tis\_getspecific

Obtains the data associated with the specified key.

### Syntax

```
tis_getspecific(
    key);
```

Argument	Data Type	Access
key	opaque pthread_key_t	read

### C Binding

```
#include <tis.h>

void *
tis_getspecific (
    pthread_key_t  key);
```

### Arguments

#### key

*key* identifies a value returned by a call to `tis_key_create`. This routine returns the data value associated with the key.

### Description

This function returns the value currently bound to the specified *key*.

This routine may be called from a data destructor function.

When threads are present, the data and keys are thread-specific; they enable a library to maintain context on a per-thread basis.

### Return Values

No errors are returned. The function `tis_getspecific` returns the data value associated with the given *key*. If no data value is associated with *key*, or if *key* is not defined, then a NULL value is returned.

### Associated Routines

```
tis_key_create
tis_key_delete
tis_setspecific
```



## tis\_key\_create

---

### tis\_key\_create

Generates a unique thread-specific data key.

#### Syntax

```
tis_key_create(  
    key,  
    destructor );
```

Argument	Data Type	Access
key	opaque pthread_key_t	write
destructor	procedure	read

#### C Binding

```
#include <tis.h>  
  
int  
tis_key_create (  
    pthread_key_t *key,  
    void (*destructor)(void *));
```

#### Arguments

##### key

Address of a variable which will receive the key value. This value is used in calls to `tis_getspecific` and `tis_setspecific` to get and set the value associated with this key.

##### destructor

Address of a procedure which is called to destroy the context value when a thread terminates with a non-NULL value for the key. Note that this argument is only utilized when threads are present.

#### Description

This routine generates a unique data key. The variable *key* provided by this routine is an opaque object used to locate data.

This routine generates and returns a new key value. The key reserves a cell. Each call to this routine creates a new cell that is unique within an application invocation. Keys must be generated from initialization code that is guaranteed to be called only once within each process. (See the `tis_once` description for more information.)

An optional destructor function may be associated with each key. At thread exit, if a key has a non-NULL destructor pointer, and the thread has a non-NULL value associated with that key, the function pointed to is called with the current associated value as its sole argument. The order in which data destructors are called at thread termination is undefined.



When threads are present, keys and any corresponding data are thread-specific; they enable the context to be maintained on a per-thread basis. For more information concerning the use of `tis_key_create` in a threaded environment, refer to the `pthread_key_create` description.

## Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacked the necessary resources to create another thread-specific data key, or the system imposed limit on the total number of keys per process {PTHREAD_KEYS_MAX} has been exceeded.
[ENOMEM]	Insufficient memory exists to create the key.
[EINVAL]	Invalid argument.

## Associated Routines

`tis_getspecific`  
`tis_key_delete`  
`tis_setspecific`  
`tis_once`



## tis\_key\_delete

---

### tis\_key\_delete

Deletes a data key.

#### Syntax

```
tis_key_delete(  
    key );
```

Argument	Data Type	Access
key	opaque pthread_key_t	write

#### C Binding

```
#include <tis.h>  
  
int  
tis_key_delete (  
    pthread_key_t  key);
```

#### Arguments

**key**  
Context key to be deleted.

#### Description

This routine deletes a data key previously returned by `tis_key_create()`. The data values associated with key need not be NULL at the time this routine is called. The application must free any application storage or perform any cleanup actions for data structures related to the deleted key or associated data. This cleanup can be done before or after this routine call. Do not attempt to use the key after calling this routine since this results in unpredictable behavior.

No destructor functions are invoked by this routine. Any destructor functions that may have been associated with key, shall no longer be called upon thread exit. `tis_key_delete` can be called from within destructor functions.

#### Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The key value is invalid.

#### Associated Routines

tis\_getspecific  
tis\_key\_create  
tis\_setspecific



## tis\_lock\_global

Locks the global mutex.

### Syntax

```
tis_lock_global( );
```

### C Binding

```
#include <tis.h>

int
tis_lock_global (void);
```

### Arguments

None

### Description

This routine locks the global mutex. The global mutex is recursive. For example, if you called `tis_lock_global` three times, `tis_unlock_global` unlocks the global mutex when you call it the third time.

For more information about actions when threads are present, refer to the `pthread_lock_global_np` description.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.

### Associated Routines

`tis_unlock_global`



## tis\_mutex\_destroy

---

## tis\_mutex\_destroy

Destroys a mutex.

### Syntax

```
tis_mutex_destroy(  
    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	write

### C Binding

```
#include <tis.h>  
  
int  
tis_mutex_destroy (  
    pthread_mutex_t  *mutex);
```

### Arguments

#### mutex

Address of the mutex (passed by reference) to be destroyed.

### Description

This routine destroys a mutex by uninitialized it, and should be called when a mutex object is no longer referenced. This routine may reclaim internal storage used by the mutex object.

It is safe to destroy an initialized mutex that is unlocked. However, it is illegal to destroy a locked mutex.

The results of this routine are unpredictable if the mutex object specified in the *mutex* argument does not currently exist, or is not initialized.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EBUSY]	An attempt is made to destroy the object referenced by <i>mutex</i> while it is locked or referenced.
[EINVAL]	The value specified by <i>mutex</i> is invalid.
[EPERM]	The caller does not have the privilege to perform the operation.



## Associated Routines

tis\_mutex\_init  
 tis\_mutex\_lock  
 tis\_mutex\_trylock  
 tis\_mutex\_unlock



## tis\_mutex\_init

Initializes a mutex.

### Syntax

```
tis_mutex_init(  
    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	write

### C Binding

```
#include <tis.h>

int
tis_mutex_init (
    pthread_mutex_t  *mutex );
```

### Arguments

**mutex**

Pointer to a mutex (passed by reference) which is initialized.

### Description

This routine initializes a mutex with the default mutex attributes. A mutex is a synchronization object that allows multiple threads to serialize their access to shared data.

The mutex is initialized and set to the unlocked state. Mutexes can be allocated in heap or static memory but not on a stack.

The `PTHREAD_MUTEX_INITIALIZER` macro can be used to statically initialize a mutex without calling this routine. Statically initialized mutexes need not be destroyed using `tis_mutex_destroy`. Use this macro as follows:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error, the mutex is not initialized, and the contents of *mutex* are undefined. Possible return values are as follows:

Return	Description
0	Successful completion.
[EAGAIN]	The system lacks the necessary resources to initialize a mutex.
[ENOMEM]	Insufficient memory exists to initialize the mutex.



Return	Description
[EBUSY]	The implementation has detected an attempt to reinitialize the <i>mutex</i> (a previously initialized, but not yet destroyed mutex).
[EINVAL]	The value specified by <i>mutex</i> is invalid.
[EPERM]	The caller does not have the privileges to perform this operation.

### Associated Routines

tis\_mutex\_init  
 tis\_mutex\_lock  
 tis\_mutex\_trylock  
 tis\_mutex\_unlock



## tis\_mutex\_lock

---

## tis\_mutex\_lock

Locks an unlocked mutex.

### Syntax

```
tis_mutex_lock(  
    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

### C Binding

```
#include <tis.h>  
  
int  
tis_mutex_lock (  
    pthread_mutex_t  *mutex);
```

### Arguments

#### mutex

Address of the mutex (passed by reference) to be locked.

### Description

This routine locks a mutex. A deadlock can result if the current owner of a mutex calls this routine in an attempt to lock the mutex a second time. (The deadlock is not detected or reported.)

In a threaded environment, the thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the mutex in the locked state and with the current thread as the mutex's current owner.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>mutex</i> is invalid.
[EDEADLK]	A deadlock condition is detected.

### Associated Routines

```
tis_mutex_destroy  
tis_mutex_init  
tis_mutex_trylock  
tis_mutex_unlock
```



## tis\_mutex\_trylock

Tries to lock a mutex.

### Syntax

```
tis_mutex_trylock(  
    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

### C Binding

```
#include <tis.h>  
  
int  
tis_mutex_trylock (  
    pthread_mutex_t  *mutex);
```

### Arguments

#### mutex

Address of the mutex (passed by reference) to be locked.

### Description

This routine tries to lock a mutex. When this routine is called, an attempt is made to immediately lock the mutex. If the mutex is successfully locked, 0 is returned. If the specified mutex is locked when this routine is called, the caller does not wait for the mutex to become available. EBUSY is returned and the thread does not wait to acquire the lock.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EBUSY]	The mutex is already locked; therefore, it was not acquired.
[EINVAL]	The value specified by <i>mutex</i> is invalid.

### Associated Routines

```
tis_mutex_destroy  
tis_mutex_init  
tis_mutex_lock  
tis_mutex_unlock
```



---

## tis\_mutex\_unlock

Unlocks a mutex.

### Syntax

```
tis_mutex_unlock(  
    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

### C Binding

```
#include <tis.h>  
  
int  
tis_mutex_unlock (  
    pthread_mutex_t *mutex);
```

### Arguments

#### **mutex**

Address of the mutex (passed by reference) which is to be unlocked.

### Description

This routine unlocks a mutex.

For more information about actions when threads are present, refer to the pthread\_mutex\_unlock description.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>mutex</i> is invalid.
[EPERM]	The caller does not own the mutex.

### Associated Routines

tis\_mutex\_destroy  
tis\_mutex\_init  
tis\_mutex\_lock  
tis\_mutex\_trylock



---

## tis\_once

Calls a one-time initialization routine that can be executed by only one thread, once.

### Syntax

```
tis_once(
    once_control,
    init_routine );
```

Argument	Data Type	Access
once_control	opaque pthread_once_t	modify
init_routine	procedure	read

### C Binding

```
#include <tis.h>

int
tis_once (
    pthread_once_t  *once_control,
    void  (*init_routine) (void));
```

### Arguments

#### once\_control

Address of a record (control block) that defines the one-time initialization code. Each one-time initialization routine in static storage must have its own unique pthread\_once\_t record.

#### init\_routine

Address of a procedure that performs the initialization. This routine is called only once, regardless of the number of times it and its associated *once\_control* are passed to *tis\_once*.

### Description

The first call to this routine by a process with a given *once\_control* will call the *init\_routine()* with no arguments. Then subsequent calls to *tis\_once()* with the same *once\_control* will not call the *init\_routine()*. On return from *tis\_once()*, it is guaranteed that the initialization routine has completed.

For example, a mutex or a context key must be created exactly once. In a threaded environment, calling *tis\_once* ensures that the initialization is serialized across multiple threads.

The *once\_control* variable must be statically initialized using the *PTHREAD\_ONCE\_INIT* macro or by zeroing out the entire structure.

---

#### Note

If you specify an *init\_routine* that directly or indirectly results in a recursive call to *tis\_once* specifying the same *init\_block* argument, the



recursive call will result in a deadlock.

---

The `PTHREAD_ONCE_INIT` macro is defined in the `tis.h` header file. A *once\_control* record must be declared as follows:

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

Note that it is often easier to simply lock a statically initialized mutex, check a control flag, and perform necessary initialization (in-line) rather than using `tis_once`. For example, code an init routine beginning with the following basic logic:

```
init()
{
    static pthread_mutex_t mutex = PTHREAD_MUTEX_INIT;
    static int flag = FALSE;

    tis_mutex_lock(&mutex);
    if(!flag)
    {
        flag = TRUE;
        /* initialize code */
    }
    tis_mutex_unlock(&mutex);
}
```

## Return Values

If an error occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	Invalid argument.



---

## tis\_raise

Sends a signal to the thread that calls it.

### Syntax

```
tis_raise(
    signal );
```

Argument	Data Type	Access
signal	integer	read

### C Binding

```
#include <tis.h>
#include <sys/signal.h>

int
tis_raise (
    int signal);
```

### Arguments

**signal**  
Specifies the signal number.

### Description

The `tis_raise()` routine sends the signal specified by the `signal` parameter to the thread that called `tis_raise` in the program. This routine is equivalent to the following: `error = pthread_kill(pthread_self(), signal);`

For related information, refer to the `pthread_kill` description.

### Return Values

Upon successful completion of the `tis_raise()` function, a value of 0 (zero) is returned. If an error condition occurs, this routine returns a nonzero value that indicates the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value of <i>signal</i> is an invalid or unsupported signal number.

### Associated Routines

`pthread_kill`  
`sigaction`



## tis\_read\_lock

---

## tis\_read\_lock

Acquires a readers/writer lock in read access mode.

### Syntax

```
tis_read_lock(  
    lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t *	write

### C Binding

```
#include <tis.h>  
  
int  
tis_read_lock (  
    tis_rwlock_t    *lock);
```

### Arguments

**lock**  
Address of the readers/writer lock.

### Description

This routine acquires a readers/writer lock in read access mode. This routine waits for any existing write access mode lock holder to relinquish its lock before granting the lock in read access mode. It returns when the lock is acquired. If the lock is already held in read access mode, the lock is granted.

Note that the type `tis_rwlock_p` is a pointer to type `tis_rwlock_t`.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>lock</i> is invalid.

### Associated Routines

```
tis_read_trylock  
tis_read_unlock  
tis_rwlock_destroy  
tis_rwlock_init  
tis_write_lock  
tis_write_trylock  
tis_write_unlock
```



## tis\_read\_trylock

Tries to acquire a readers/writer lock in read access mode. Does not wait if the lock cannot be immediately granted.

### Syntax

```
tis_read_trylock(  
    lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

### C Binding

```
#include <tis.h>  
  
int  
tis_read_trylock (  
    opaque tis_rwlock_t    *lock);
```

### Arguments

**lock**  
Address of the readers/writer lock.

### Description

This routine tries to acquire a readers/writer lock in read access mode. If the lock cannot be granted, the routine returns without waiting. When a thread calls this routine, an attempt is made to immediately acquire the lock in read mode. If the lock is acquired, 0 is returned. If a write access mode lock holder exists, EBUSY is returned. If the lock cannot be obtained immediately, the calling program does not wait for the lock to be released.



## tis\_read\_trylock

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion; the lock was acquired.
[EBUSY]	The lock is being held with write access mode. The lock was not acquired.

### Associated Routines

- tis\_read\_lock
- tis\_read\_unlock
- tis\_rwlock\_destroy
- tis\_rwlock\_init
- tis\_write\_lock
- tis\_write\_trylock
- tis\_write\_unlock



---

## tis\_read\_unlock

Unlocks a readers/writer lock that was acquired in read access mode.

### Syntax

```
tis_read_unlock(  
    lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

### C Binding

```
#include <tis.h>  
  
int  
tis_read_unlock (  
    tis_rwlock_t  *lock);
```

### Arguments

**lock**  
Address of the readers/writer lock.

### Description

This routine unlocks a readers/writer lock that was acquired in read access mode. If there are no other holders of the lock with read access mode and another thread is waiting to acquire the lock in write access mode, the lock will now be granted.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>lock</i> is invalid.

### Associated Routines

```
tis_read_lock  
tis_read_trylock  
tis_rwlock_destroy  
tis_rwlock_init  
tis_write_lock  
tis_write_trylock  
tis_write_unlock
```



## tis\_rwlock\_destroy

---

## tis\_rwlock\_destroy

Destroys a readers/writer lock.

### Syntax

```
tis_rwlock_destroy(  
    lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

### C Binding

```
#include <tis.h>  
  
int  
tis_rwlock_destroy (  
    tis_rwlock_t *lock);
```

### Arguments

#### lock

Address of the readers/writer lock structure to be destroyed.

### Description

This routine destroys a readers/writer lock. The routine has the reverse operation of the `tis_rwlock_init` routine which began the session by initializing the `tis_rwlock_t` structure. Ensure that there are no locks granted, or threads waiting for locks to be granted, prior to calling this routine. This routine should be called only after the readers and writers are done using the lock.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EBUSY]	The lock is in use.

### Associated Routines

```
tis_read_lock  
tis_read_trylock  
tis_read_unlock  
tis_rwlock_init  
tis_write_lock  
tis_write_trylock  
tis_write_unlock
```



## tis\_rwlock\_init

Initializes readers/writer lock.

### Syntax

```
tis_rwlock_init(
    lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

### C Binding

```
#include <tis.h>

int
tis_rwlock_init (
    tis_rwlock_t  *lock);
```

### Arguments

#### lock

Address of a readers/writer lock's structure.

### Description

This routine should be called to initialize a readers/writer lock. The routine initializes the `tis_rwlock_t` structure that holds the lock states. To destroy a lock, call the `tis_rwlock_destroy` routine.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>lock</i> is invalid.
[ENOMEM]	Insufficient memory exists to establish lock.

### Associated Routines

```
tis_read_lock
tis_read_trylock
tis_read_unlock
tis_rwlock_destroy
tis_write_lock
tis_write_trylock
tis_write_unlock
```



---

## tis\_self

Obtains the identifier of the current thread.

### Syntax

```
tis_self(  
    void);
```

### C Binding

```
#include <tis.h>  
  
pthread_t  
tis_self(void);
```

### Arguments

None

### Description

This routine allows a thread to obtain its own thread identifier.

This value becomes meaningless when the thread is deleted.

The initial thread in a process may "change identity" when thread system initialization completes (that is, when the DECthreads library is loaded).

### Return Values

Returns the thread identifier of the caller.

### Associated Routines

`pthread_create`



---

## tis\_setcancelstate

Sets the caller's cancelability state.

### Syntax

```
tis_setcancelstate(
    state,
    oldstate );
```

Argument	Data Type	Access
state	integer	read
oldstate	integer	write

### C Binding

```
#include <tis.h>

int
tis_setcancelstate (
    int    state,
    int    *oldstate );
```

### Arguments

#### state

State of general cancelability to set for the calling thread. Valid state values are as follows:

- PTHREAD\_CANCEL\_ENABLE
- PTHREAD\_CANCEL\_DISABLE

#### oldstate

Previous cancelability state.

### Description

This routine sets the caller's cancelability to *state* and returns the previous cancelability state to the location referenced by *oldstate*.

When cancelability state is set to PTHREAD\_CANCEL\_DISABLE, a cancel cannot be delivered to the thread, even if a cancelable routine is called or asynchronous cancelability type is enabled.

When a thread is created, the default general cancelability state is PTHREAD\_CANCEL\_ENABLE. When *tis\_setcancelstate* is called prior to loading threads, the cancel state propagates to the initial thread in the executing program.



### Possible Problems When Disabling Cancelability

The most important use of cancel calls is to ensure that indefinite wait operations are terminated. For example, a thread waiting on some network connection, which may take days to respond (or may never respond), should be made cancelable.

When cancelability is disabled, no routine is cancelable. As a result, the user is unable to cancel the operation. When disabling cancelability, be sure that no long waits can occur or that it is necessary for other reasons to defer cancels around that particular region of code.

### Return Values

On successful completion, this routine returns the previous state of general cancelability.

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The specified state is not PTHREAD_CANCEL_ENABLE or PTHREAD_CANCEL_DISABLE.

### Associated Routines

tis\_testcancel



## tis\_setspecific

Sets the data value associated with the specified key.

### Syntax

```
tis_setspecific(
    key,
    value );
```

Argument	Data Type	Access
key	opaque pthread_key_t	read
value	void *	read

### C Binding

```
#include <tis.h>

int
tis_setspecific (
    pthread_key_t key,
    const void *value);
```

### Arguments

#### key

Key that identifies the data to receive *value*. This key value must be obtained from `tis_key_create`.

#### value

New data value to associate with the specified key. Once set, this value can be retrieved using the same key in a call to `tis_getspecific`.

### Description

This routine sets the data value associated with the specified key. If a value is defined for the key (the current value is not NULL), the new value is substituted for it. The key is obtained by a previous call to `tis_key_create`.

This routine may be called from a data destructor function. However, calling this routine from a destructor may result in lost storage or infinite loops.

### Return Values

If an error condition occurs, this routine returns an integer indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The key value is invalid.



## tis\_setspecific

Return	Description
[ENOMEM]	Insufficient memory exists to associate the value with the key.

### Associated Routines

tis\_getspecific  
tis\_key\_create  
tis\_key\_delete



---

## tis\_testcancel

Creates a cancellation point in the current thread.

### Syntax

```
tis_testcancel();
```

### C Binding

```
#include <tis.h>

void
tis_testcancel (void);
```

### Arguments

None

### Description

This routine creates a cancellation point in the calling thread. The cancel is delivered only if a cancel is pending for the current thread and the cancelability state is enabled. (A thread disables delivery of cancels to itself by calling `tis_setcancelstate`.)

This routine, when called within very long loops, ensures that a pending cancel is noticed within a reasonable amount of time.

### Return Values

None

### Associated Routines

`tis_setcancelstate`



## tis\_unlock\_global

---

### tis\_unlock\_global

Unlocks the global mutex.

#### Syntax

```
tis_unlock_global( );
```

#### C Binding

```
#include <tis.h>

int
tis_unlock_global (void);
```

#### Arguments

None

#### Description

This routine unlocks the global mutex. Since the global mutex is recursive, the unlock will occur when each call to `tis_lock_global` has been matched by a call to this routine. For example, if you called `tis_lock_global` three times, `tis_unlock_global` unlocks the global mutex when you call it the third time.

For more information about actions when threads are present, refer to the `pthread_unlock_global_np` description.

#### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EPERM]	The global mutex is unlocked or locked by another thread.

#### Associated Routines

`tis_lock_global`



## tis\_write\_lock

Acquires a readers/writer lock in write access mode.

### Syntax

```
tis_write_lock(  
    lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

### C Binding

```
#include <tis.h>  
  
int  
tis_rwlock (   
    tis_rwlock_t    *lock);
```

### Arguments

#### lock

Address of the readers/writer lock.

### Description

This routine acquires a readers/writer lock in write access mode. This routine waits for any other active locks (in either read access or write access mode) to become unlocked before the lock request is granted. The routine returns when the readers/writer lock is established.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>lock</i> is invalid.

### Associated Routines

```
tis_read_lock  
tis_read_trylock  
tis_read_unlock  
tis_rwlock_destroy  
tis_rwlock_init  
tis_write_trylock  
tis_write_unlock
```



## tis\_write\_trylock

---

## tis\_write\_trylock

Tries to acquire a readers/writer lock in write access mode.

### Syntax

```
tis_write_trylock(  
    lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

### C Binding

```
#include <tis.h>  
  
int  
tis_write_trylock (  
    tis_rwlock_t    *lock);
```

### Arguments

**lock**  
Address of the readers/writer lock.

### Description

This routine tries to acquire a readers/writer lock in write access mode. An attempt is made to immediately acquire the lock. If the lock is acquired, 0 is returned. If the lock is held by another thread (in either read access mode or write access mode), EBUSY is returned and the caller does not wait for an eventual lock. Note that if the lock is already held by the thread attempting to acquire it in write access mode, that would be a coding error (EBUSY will be returned anyway however, as no ownership error checking is done.)

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type of error. Possible return values are as follows:

Return	Description
0	Successful completion; the write lock is acquired.
[EBUSY]	The lock in write access mode was NOT acquired, as it is already held by another thread.



## Associated Routines

tis\_read\_lock  
 tis\_read\_trylock  
 tis\_read\_unlock  
 tis\_rwlock\_destroy  
 tis\_rwlock\_init  
 tis\_write\_lock  
 tis\_write\_unlock



## tis\_write\_unlock

---

## tis\_write\_unlock

Unlocks a readers/writer lock.

### Syntax

```
tis_write_unlock(  
    lock );
```

Argument	Data Type	Access
lock	opaque tis_rwlock_t	write

### C Binding

```
#include <tis.h>  
  
int  
tis_write_unlock (  
    tis_rwlock_t  *lock);
```

### Arguments

**lock**  
Address of the readers/writer lock.

### Description

This routine unlocks a readers/writer lock that was acquired in write access mode. Upon completion of this routine, any thread waiting to acquire the lock in read access mode will have the locks granted. If no threads are waiting to acquire the lock in read access mode, then a thread waiting to acquire it in write access mode will have the lock granted.

### Return Values

If an error condition occurs, this routine returns an integer value indicating the type error. Possible return values are as follows:

Return	Description
0	Successful completion.
[EINVAL]	The value specified by <i>lock</i> is invalid.

### Associated Routines

tis\_read\_lock  
tis\_read\_trylock  
tis\_read\_unlock  
tis\_rwlock\_init  
tis\_rwlock\_destroy  
tis\_write\_lock  
tis\_write\_trylock



# Part IV

---

## Appendices

Part IV contains appendices that provide supporting information about DECthreads, such as operating system-specific information, debugging information, and additional reference information.



# Part IV

## Appendices

Appendix A: List of abbreviations and symbols used in the text.  
Appendix B: List of abbreviations and symbols used in the figures.  
Appendix C: List of abbreviations and symbols used in the tables.



---

## Considerations for Digital UNIX Systems

This appendix discusses DECthreads issues specific to systems based on the Digital UNIX operating system.

### A.1 Overview

DECthreads contains four application programming interfaces (APIs):

- **POSIX 1003.1c Standard API**

With Version 4.0 of the Digital UNIX operating system, the DECthreads library (PTHREAD\$RTL.EXE) implements the POSIX 1003.1c standard interface, as approved by the IEEE standards board in June 1995 (IEEE Std 1003.1c-1995, POSIX System Application Program Interface). The new POSIX (pthread) interface supported with DECthreads is Digital UNIX's most portable, efficient, and powerful programming interface for a multithreaded environment. These interfaces are defined by the C language header file "pthread.h".

- **tis**

Version 4.0 of the Digital UNIX operating system includes the Thread Independent Services (tis) application programming interface (CMA\$TIS\_SHR.EXE). tis provides services that assist with the development of thread-safe APIs. Thread synchronization can involve significant run-time cost, which is undesirable in the absence of threads. tis enables you to build thread-safe APIs that are efficient in the nonthreaded environment, yet provide the necessary synchronization in the threaded environment. When DECthreads is not active within the process, tis executes only the minimum steps necessary: code running in a nonthreaded environment is not burdened by the run-time synchronization that is necessary when the same code is run in a threaded environment. When DECthreads is active, the tis functions provide the necessary synchronization for thread safety.

- **cma**

The cma interface of DECthreads will be obsolete in a future release. Obsolescence means that this API will always exist in the Digital UNIX operating system and will be supported, but will no longer be documented or enhanced. It is recommended that you port your cma-based application to the IEEE Std 1003.1c-1995, POSIX System Application Program Interface provided by DECthreads.

- **POSIX 1003.4a Draft 4 API**

Two variants of the POSIX 1003.4a Draft 4 interface (a very early draft of POSIX 1003.1c) were supplied with earlier versions of DECthreads. One variant directly supported the Posix 1003.4a Draft 4 interfaces, and the other modified them by raising exceptions to report errors (as does the cma interface).



Both of these interfaces are being retired, and will be removed in a future release. Programs using these interfaces should be modified to use the IEEE Std 1003.1c-1995, POSIX System Application Program Interface provided by DECthreads. A compatibility mode for the POSIX 1003.4a Draft 4 APIs has been provided in this release to help ease migration. This compatibility mode will be removed in a future release.

## A.2 Digital UNIX Systems

The Digital UNIX operating system supports multiple concurrent streams of execution within a process using the Mach kernel. DECthreads utilizes these kernel execution contexts to implement user threads. One important benefit of this is that user threads can run simultaneously on separate processors in a multiprocessor system.

On a system with kernel threads (especially on a multiprocessor system), it is critical to remember that thread scheduling policy and priority cannot be substituted for synchronization. High-priority and low-priority threads may run at the same time on different processors, or may be switched by the operating system beyond the control of DECthreads. See Section 3.1 to be sure your application works correctly on a multiprocessor system.

### A.2.1 Including DECthreads Header Files

Include one of the DECthreads header files shown in Table A-1 in your program to use the appropriate DECthreads library:

**Table A-1 DECthreads Header Files**

Header File	Interface
pthread.h	POSIX 1003.1c-1995 routines
tis.h	tis routines (tis_)
cma.h	cma routines (cma_)
pthread_d4.h	POSIX 1003.4a Draft 4 routines with status-returning interface
pthread_exc.h	POSIX 1003.4a Draft 4 routines with exception-returning interface

Do not include more than one of the above header files in your module.

### A.2.2 Compiling Multithreaded Applications Libraries

Multithreaded applications are compiled using shared libraries. For a discussion about shared libraries, see the *Digital UNIX Programmer's Guide*.

Table A-2 contains the libraries supported for multithreaded programming.

**Table A-2 Digital UNIX Shared Libraries for Multithreaded Programs**

libmach.so	Shared version of threads support library. Direct use of mach interfaces is not supported.
------------	--

(continued on next page)



**Table A-2 (Cont.) Digital UNIX Shared Libraries for Multithreaded Programs**

libpthread.so	Shared version of the base pthreads package. Requires libmach.so, libexc.so, and libc.so
libexc.so	Shared version of Digital UNIX exception support package.
libpthreads.so	Shared version of DECthreads "legacy" package, implementing the cma and 1003.4a/Draft 4 API sets.
libc.so	Shared version of libc package. (C language runtime.)

Compile a multithreaded application using shared versions of libexc, libmach, libpthread, and libc as follows:

```
% cc -o myprog myprog.c -pthread
```

### **A.2.3 Compiling Applications That Use the Draft 4 POSIX 1003.4a Standard Interface**

The POSIX 1003.4a, Draft 4 interface of DECthreads is being retired in a future release. Applications that were written using the POSIX 1003.4a, Draft 4 API should be migrated to the new IEEE Std 1003.1c-1995, POSIX System Application Program Interface provided by DECthreads. A compatibility mode for the draft 4 POSIX 1003.4a API has been provided in this release to help ease migration. This compatibility mode will be removed in a future release. Applications utilizing the compatibility mode should be compiled using the following:

```
% cc -o myprog myprog.c -threads
```

This will include the proper pthread\_d4.h file for you with no source file modification.

### **A.2.4 Linking Multithreaded Shared Libraries**

The `ld` command does not support the `-pthread` or `-threads` switches. Instead, you must list the individual libraries in the proper order.

For libraries using only the 1003.1c-1995 interface, you would use the following:

```
ld <...> -lpthread -lmach -lexc -lc
```

If using the cma or 1003.42 Draft 4 interfaces, you would use the following:

```
ld <...> -lpthreads -lpthread \ -lmach -lexc -lc
```

### **A.2.5 Compiling Applications That Use the Thread-Independent Services (tis) Interface**

Applications that use the tis interface should include the `<tis.h>` header file and link against `libc.so`.

### **A.2.6 Support for Real-Time Scheduling on Digital UNIX Systems**

DECthreads supports Digital UNIX real-time scheduling. This allows you to set the scheduling policy and priority of threads. By default, threads are created using process contention scope. This means that the full range of 1003.1c-1995 scheduling policy and priority is available. However, threads running in process contention scope do not preempt lower priority threads in another process. For example, a thread in process contention scope in `SCHED_FIFO` policy with `PRI_FIFO_MAX` priority will not preempt a thread in another process running in `SCHED_FIFO` with `PRI_FIFO_MIN`.



## A.2.7 Thread Cancelability of System Services

Blocking system calls are cancellation points when called in a thread created using the POSIX 1003.1c `pthread_create` function. Threads created using the cma or draft 4 POSIX standard interfaces disable this capability to preserve binary compatibility. None of the system calls should be called with asynchronous cancellation enabled. For more information, see Section 2.6.

## A.3 Using Signals

This section discusses the types of signals, POSIX 1003.1c-1995 signal handling, and alternatives to using signals.

### A.3.1 Types of Signals

Signals are delivered as a result of some event. UNIX signals are grouped into the following four categories of pairs that are orthogonal to each other:

- Terminating and synchronous
- Nonterminating and asynchronous
- Nonterminating and synchronous
- Terminating and asynchronous

The action taken when a particular signal is delivered depends on the characteristics of that signal.

#### A.3.1.1 Nonterminating Signals

**Nonterminating** signals do not result in the termination of the process by default.

Nonterminating signals represent events that can be either internal or external to the process. The process might desire notification about or ignore these events. When a nonterminating asynchronous signal is delivered to the process, any thread waiting for the signal with `sigwait(2)` is awakened. If a signal handler is installed for the signal it will be run in a single thread.

#### A.3.1.2 Terminating Signals

**Terminating** signals result in the termination of the process by default. Whether a particular signal is terminating or not is independent of whether it is synchronously or asynchronously delivered.

#### A.3.1.3 Asynchronous Signals

**Asynchronous** signals are the result of an event that is external to the process and are delivered at any point in a thread's execution when such an event occurs. For example, when a user running a program types the interrupt character at the terminal (generally Ctrl/C), a SIGINT (interrupt signal) is delivered to the process.

Asynchronous, terminating signals represent an occurrence of an event that is external to the process, and, if unhandled, results in the termination of the process. If a thread is waiting, using `sigwait(2)`, it is awakened, and the signal is dismissed. If a signal handler is installed, it will be called in a single (arbitrary) thread. Otherwise, the process is terminated.



#### A.3.1.4 Synchronous Signals

**Synchronous** signals are the result of an event that occurs inside a process and are delivered synchronously with respect to that event. For example, if a floating-point calculation results in an overflow, then a SIGFPE (floating-point exception signal) is delivered to the process immediately following the instruction that resulted in the overflow.

Synchronous, terminating signals represent an error that has occurred in the currently executing thread. For more information, see Section A.4.

#### A.3.2 POSIX sigwait Service

The POSIX 1003.1c sigwait service allows any thread to block until one of a specified set of signals is delivered. A thread waits for any of the asynchronous signals except for SIGKILL and SIGSTOP.

A thread should not wait for a synchronous signal. This is because synchronous signals are the result of an error during the execution of a thread, and if the thread is waiting for a signal, then it is not executing. Therefore, a synchronous signal cannot occur for a particular thread while it is waiting, and so the thread will wait forever.

POSIX stipulates that the thread must block the signals it will wait for before calling sigwait.

#### A.3.3 Signal Alternatives Using the sigwait Routine

Avoid dealing with UNIX signals directly by using signal handler routines in multithreaded programs. POSIX 1003.1c-1995 provides alternatives to signal handling.

One alternative to using asynchronous signals directly is to use the sigwait() routine. The sigwait() routine takes a signal mask (POSIX sigset\_t type) as an argument and returns the number of a signal (int) in the second argument. The sigwait() routine causes the calling thread to block (without affecting other threads) until one of the signals in the sigset\_t is received. The routine will then return with the number of that signal in the second argument.

For example, you can create a thread that blocks on a sigwait() routine for SIGINT, rather than handling a Ctrl/C in the normal way. This thread could then alert (cancel) other threads to cause the program to shut down the current activities.

Following are two reasons for avoiding signals:

- Signals cannot be used in a modular way in a multithreaded program.
- Signals, used as an asynchronous programming technique, are unnecessary in a multithreaded program.

In a multithreaded program, signals cannot be used in a modular way because there is only one signal handler routine for all of the threads in an application. If two threads install different signal handlers for the signal, all threads will dispatch to the last handler when they receive the signal.

Do not use asynchronous programming techniques in conjunction with threads, particularly those that are intended to increase parallelism, such as using timer signals and I/O signals. These techniques are complicated and error-prone; they are also unnecessary because threads provide a mechanism for parallel execution that is simpler and less error-prone. Furthermore, most of the threads services



are not supported for use in signal handlers, and most run-time library functions cannot be used reliably inside a signal handler.

## A.4 Signals Reported as Exceptions

The following sections list UNIX signals that are reported as DECthreads exceptions by default. If any thread declares an action for one of these signals (using `sigaction(2)` or equivalent), no thread in the process can receive the exception.

### A.4.1 Synchronous Terminating Signals

Table A-3 shows the UNIX signals that are considered synchronous terminating and the DECthreads exceptions that are associated with those signals.

**Table A-3 Synchronous Terminating Signals**

Signal	Exception
SIGILL	<code>pthread_exc_illinstr_e</code>
SIGIOT	<code>pthread_exc_SIGIOT_e</code>
SIGEMT	<code>pthread_exc_SIGEMT_e</code>
SIGFPE	<code>pthread_exc_aritherr_e</code>
SIGBUS	<code>pthread_exc_illaddr_e</code>
SIGSEGV	<code>pthread_exc_illaddr_e</code>
SIGSYS	<code>pthread_exc_SIGSYS_e</code>
SIGPIPE	<code>pthread_exc_SIGPIPE_e</code>

## A.5 Dynamic Activation

Dynamic activation of DECthreads (or code that depends on DECthreads) is currently not supported.



## Considerations for OpenVMS Systems

This appendix discusses DECthreads issues and restrictions specific to the OpenVMS operating system.

### B.1 Overview

DECthreads contains four application programming interfaces (APIs):

- **POSIX 1003.1c Standard API**

With Version 7.0 of the OpenVMS operating system, the DECthreads library (PTHREAD\$RTL.EXE) implements the POSIX 1003.1c standard interface, as approved by the IEEE standards board in June 1995 (IEEE Std 1003.1c-1995, POSIX System Application Program Interface). The new POSIX (pthread) interface supported with DECthreads is OpenVMS's most portable, efficient, and powerful programming interface for a multithreaded environment. These interfaces are defined by the C language header file "pthread.h".

- **tis**

The OpenVMS Version 7.0 operating system includes the Thread Independent Services (tis) application programming interface (CMA\$TIS\_SHR.EXE). tis provides services that assist with the development of thread-safe APIs. Thread synchronization can involve significant run-time cost, which is undesirable in the absence of threads. tis enables you to build thread-safe APIs that are efficient in the nonthreaded environment, yet provide the necessary synchronization in the threaded environment. When DECthreads is not active within the process, tis executes only the minimum steps necessary: code running in a nonthreaded environment is not burdened by the run-time synchronization that is necessary when the same code is run in a threaded environment. When DECthreads is active, the tis functions provide the necessary synchronization for thread safety.

- **cma**

The cma interface of DECthreads will be obsolete in a future release. Obsolescence means that this API will always exist in the OpenVMS operating system and will be supported, but will no longer be documented or enhanced. It is recommended that you port your cma-based application to the IEEE Std 1003.1c-1995, POSIX System Application Program Interface provided by DECthreads.

- **POSIX 1003.4a Draft 4 API**

Two variants of the POSIX 1003.4a Draft 4 interface (a very early draft of POSIX 1003.1c) were supplied with earlier versions of DECthreads. One variant directly supported the Posix 1003.4a Draft 4 interfaces, and the other modified them by raising exceptions to report errors (as does the cma interface).



Both of these interfaces are being retired, and will be removed in a future release. Programs using these interfaces should be modified to use the IEEE Std 1003.1c-1995, POSIX System Application Program Interface provided by DECthreads. A compatibility mode for the POSIX 1003.4a Draft 4 APIs has been provided in this release to help ease migration. This compatibility mode will be removed in a future release.

## B.2 Including DECthreads Header Files

The DECthreads C language header files shown in Table B-1 provide interface definitions for DECthreads APIs.

**Table B-1 DECthreads Header Files**

Header File	Interface
tis.h	tis routines (tis_)
cma.h	cma routines (cma_)
cma\$def.h	cma routines (cma\$)
pthread.h	POSIX 1003.1c-1995 routines
pthread_d4.h	POSIX 1003.4a Draft 4 routines with status-returning interface
pthread_exc.h	POSIX 1003.4a Draft 4 routines with exception-returning interface

Do not include more than one of the above header files in your module.

## B.3 Compiling OpenVMS Images

When compiling threaded applications which utilize the POSIX 1003.1c standard interface, the cma interface, or the tis interface, no special compiler definitions are required.

## B.4 Compiling OpenVMS Images Using the POSIX 1003.4a Draft 4 Standard

When compiling threaded applications in the C Language that use the POSIX 1003.4a Draft 4 standard interface or its exception handling interface, define PTHREAD\_USE\_D4 on the compiler command line.

For example:

```
cc/define=PTHREAD_USE_D4 myprog.c
```

This will include the proper header file for you (pthread\_d4.h), so that you do not need to modify your source includes.



## B.5 Linking OpenVMS Images

When you link an image that calls DECthreads routines, you must link against the appropriate images. These images are listed in Table B-2.

**Table B-2 DECthreads Images**

Image	Routine Library
pthread\$rtl.exe	POSIX 1003.1c Standard Interface
cma\$tis_shr.exe	Thread Independent Services
cma\$lib_shr.exe	CMALIB interface (CMA\$)
cma\$rtl.exe	CMA\$ interface
cma\$open_lib_shr.exe	CMALIB interface (CMA_)
cma\$open_rtl.exe	CMA_, Draft 4 and Draft 4 exception interface

PTHREAD\$RTL.EXE, CMA\$TIS\_SHR.EXE, CMA\$RTL.EXE, and CMA\$LIB\_SHR.EXE are included in the IMAGELIB library, making it unnecessary to specify those images (unless you are using the /NOSYSLIB with the linker) in a Linker options file.

When you link an image that utilizes the cma\$open\_lib\_shr.exe and cma\$open\_rtl.exe images, they must be specified in a Linker options file.

DECthreads is supplied only as shareable images. It is not supplied as object libraries.

## B.6 Using DECthreads with Asynchronous System Trap (AST) Routines

An AST is an OpenVMS mechanism for reporting an asynchronous event to a process. The following are restrictions concerning the use of ASTs with DECthreads:

- Avoid blocking ASTs using any mechanism other than \$SETAST.
- Be aware that blocking ASTs in one thread may prevent delivery of ASTs which are actually intended for other threads. Therefore, it is best to avoid blocking ASTs for an extended period of time. Also, it is best to avoid calling DECthreads functions which may block the thread while it has disabled ASTs.
- Do not call DECthreads routines, except those that have the \_int (interrupt) suffix in their names, from within an AST routine. Calling any other DECthreads routines from code running in an AST can be unreliable or cause unexpected behavior.
- For OpenVMS Alpha, ASTs are handed off to DECthreads by the operating system. This allows ASTs to be delivered in the context of the appropriate thread. On a multiprocessor machine it may be possible to have a thread executing an AST routine in parallel with another thread's execution. When a thread disables ASTs, not only does it block out its own ASTs, but it prevents delivery of any ASTs that do not specifically belong to a particular thread as well.



## B.7 Dynamic Activation

Dynamic activation of DECthreads (or images that depend on DECthreads) is currently not supported.

## B.8 Declaring an OpenVMS Condition Handler

This section discusses a restriction on declaring an OpenVMS condition handler while using DECthreads exceptions and DECthreads behavior when a condition is signalled.

The following are three ways to declare an OpenVMS condition handler:

- Calling `VAX$ESTABLISH` (from a program written in C)
- Calling `LIB$ESTABLISH`
- Placing the address of the condition handler directly into the stack frame (from a program written in VAX MACRO or VAX BLISS)

Do not declare an OpenVMS condition handler within a DECthreads `TRY/ENDTRY` exception block. Doing so deletes without notification any handler that exists for the current procedure. If your code declares a condition handler within the `TRY/ENDTRY` block, DECthreads exceptions will not be handled correctly until the next `TRY` statement is executed. The `TRY` statement restores the DECthreads condition handler.

On OpenVMS VAX, you can declare a condition handler outside of a `TRY/ENDTRY` block with no restrictions. If a condition handler has already been declared when you execute a `TRY` statement, DECthreads saves the previous handler address. When DECthreads receives a condition it does not handle (including `SS$_UNWIND`, `SS$_DEBUG` or a condition code that does not have a *SEVERE* severity), DECthreads invokes the saved condition handler. The condition handler will be re-established when the `TRY` block exits.

## B.9 Thread Cancelability of System Services

On OpenVMS Alpha, system calls are now cancellation points for threads created using the POSIX 1004.1c interface. System calls are not cancellation points for threads created using the cma or draft 4 POSIX 1003.4a interface. None of the system calls should be called with asynchronous cancellation enabled. For more information, see Section 2.6.

## B.10 Using OpenVMS Alpha 64-Bit Addressing

On OpenVMS Alpha, DECthreads supports the use of 64-bit addressing only in the POSIX 1003.1c-1995 standard interface. When compiling with `CC/POINTER_SIZE=LONG`, the function name `pthread_join` returns a 64-bit "void\*" for the result value. You can also use `pthread_join64` or `pthread_join32` to specify the desired return size. Note that no other functions have special 64-bit versions, since the OpenVMS Alpha calling standard always supports 64-bit arguments and return values.



## B.11 DECthreads Condition Values

Table B-3 lists the DECthreads condition values for OpenVMS systems and provides an explanation and user action.

Table B-3 DECthreads Condition Values

Condition Value	Explanation and User Action
CMA\$_EXCCOP	<p>Exception raised; OpenVMS condition code follows</p> <p>Explanation: One of the DECthreads exception commands (RAISE or RERAISE) raised or reraised an exception condition originating outside the DECthreads library. The secondary condition code in the signal vector will be the original code.</p> <p>User Action: See the documentation for the software that your program is calling to determine the reason for this exception.</p>
CMA\$_EXCCOPLOS	<p>Exception raised; some information lost</p> <p>Explanation: CMA\$_EXCCOPLOS is nearly the same as CMA\$_EXCCOP except that DECthreads determined that the copied signal vector may contain address arguments. However, the address arguments may not be valid when the stack is unwound and the condition is resignaled. Therefore, DECthreads clears the condition codes' arguments in the resignaled vector. In most cases, DECthreads knows that SS\$_code arguments are "safe" and will not clear them. Most other codes with arguments will result in CMA\$_EXCCOPLOS.</p> <p>User Action: See the documentation for the software that your program is calling to determine the reason for this exception.</p>
CMA\$_EXCEPTION	<p>Exception raised; address of exception object is object-address</p> <p>Explanation: This condition is used as the primary condition to RAISE an address-type DECthreads exception. The condition is signaled with a single argument containing the address of the EXCEPTION structure. There is no support for interpreting this value. It is only meaningful to the facility that defined the EXCEPTION. It is not good programming practice to let an address exception propagate outside the facility that raised it. There is no support for getting message text, and it cannot be interpreted by other facilities.</p> <p>User Action: None.</p>

## B.12 Two-Level Scheduling on OpenVMS Alpha

This section applies to OpenVMS Alpha only. Starting with OpenVMS Alpha Version 7.0, DECthreads implements a new scheduling model, referred to as **two-level scheduling**. This model is based on the concept of **virtual processors**. DECthreads schedules threads onto virtual processors similar to the way that



OpenVMS schedules processes onto the processors of a multiprocessing machine. From the DECthreads perspective, a scheduled thread is executed on a virtual processor until it blocks or until it exhausts its timeslice quantum. Then, DECthreads schedules a new thread to run. Virtual processors are implemented using the new kernel thread support in the OpenVMS operating system. While DECthreads is scheduling threads onto virtual processors, the OpenVMS scheduler is scheduling virtual processors to run on physical processors. The term "two-level" scheduling is derived from this relationship.

This division provides two major advantages:

First, it allows most thread scheduling to take place completely in User Mode, without the intervention of the OpenVMS scheduler. Since a thread context switch does not involve any privileged information (it is basically just swapping registers), it can be done much more efficiently in User Mode than a context switch involving the operating system.

Second, the two-level scheduling model allows the OpenVMS scheduler to schedule virtual processors onto separate processors of a multiprocessing machine. This allows a process using DECthreads to take advantage of the full resources of a multiprocessor machine.

The key to making the two-level scheduling model work is the **upcall**. An upcall is a communication mechanism between the OpenVMS scheduler and the DECthreads scheduler. When an event occurs that affects the scheduling of a thread, such as blocking for a system service, the OpenVMS scheduler calls "up" to the DECthreads scheduler to notify it of the change in the thread's status. This upcall gives DECthreads the opportunity to schedule another thread to run on the virtual processor in place of the blocking thread, rather than to allow the virtual processor itself to block, which would deny that resource to other threads in the process. Upcalls are typically arranged in pairs, with an "unblock" upcall corresponding to each "block" upcall. The unblock upcall notifies DECthreads that a previously blocked thread is now eligible to run again. DECthreads then schedules that thread to run point when it is appropriate, given the thread's scheduling policy and priority.

### **B.12.1 DECthreads Virtual Processors**

Virtual processors are created as they are needed by the application. The number of virtual processors that DECthreads creates is limited by the SYSGEN parameter **MULTITHREAD**. This parameter is typically set to the number of processors that reside in the system. In general, there is no reason to create more virtual processors than there are physical processors. The virtual processors would contend with each other for the physical processors and cause unnecessary overhead. Regardless of the value of the **MULTITHREAD** parameter, DECthreads will create no more virtual processors than there are user threads (excluding DECthreads internal threads).

DECthreads does not delete virtual processors or let them terminate. They are retained in an idle (HIB) state until they are needed again. During image-rundown, they are deleted by OpenVMS.

The DECthreads scheduler may schedule any user thread onto any virtual processor. Therefore, a user thread may run on different kernel threads at different times. Normally, this should pose no problem. However (for example), a user thread's PID (as retrieved by querying the system) may change from time to time.



## B.12.2 AST delivery

When a User Mode AST becomes deliverable to a DECthreads process, the OpenVMS scheduler makes an upcall to DECthreads, passing the information that is required to deliver the AST (service routine address, argument, and target user thread ID). DECthreads stores this information and queues the AST to be delivered to the appropriate user thread. That thread is made runnable (if it is not already), and executes the AST routine the next time it is scheduled to run. This means the following things:

- A per-thread AST will interrupt the user thread that requested it, regardless of which virtual processor on which the thread is running.
- The AST will be run at the priority of the target thread, so that low-priority threads' ASTs do not preempt or interfere with the execution of high-priority threads.
- The AST routine executes in the context of the target thread, so that the danger of surprise stack overflows is diminished, and stack-walks and exception propagation work as they should.

In addition to per-thread ASTs, there are also User Mode ASTs that are directed to the process as a whole, or to no thread in particular, or to a thread that has since terminated. These "process" ASTs are queued to the initial thread, making the thread runnable in a fashion similar to per-thread ASTs. They are executed in the context of the initial thread, for the following reasons:

- The initial thread has an expandable stack, unlike the other threads, so this minimizes the danger of stack space problems.
- Any code that is making assumptions about particular characteristics of AST delivery is most likely running in the initial thread, so delivering the AST to the initial thread is least likely to cause problems.
- The initial thread gets a boost to the top scheduling priority, to ensure that the process ASTs are executed promptly. Since these ASTs cannot be ascribed to any particular thread, their priority cannot be assessed, so it is important that they be delivered promptly in the event that a high-priority thread is waiting to be signalled by one of them.

---

### Note

---

In OpenVMS Version 7.0, all ASTs are directed to the process as a whole. In future releases, AST delivery will be made per thread, as individual services are updated.

---

The following implications must be considered for application development:

- If an application makes heavy use of ASTs, it may (to a certain extent) starve the initial thread, since only that thread executes the ASTs that are directed to the process as a whole (as opposed to the pre-Version 7.0 behavior of starving all threads equally).
- There are also implications for controlling AST delivery. \$SETAST generates an upcall similar to the one for AST delivery. This allows DECthreads to note the request by a thread to block (or unblock) AST delivery. When a thread has requested that ASTs be blocked, it will not receive delivery of any per-thread ASTs; nor will the process receive delivery of any process ASTs. This is, in effect, the same behavior as in pre-Version 7.0, except that



a second thread cannot undo a block requested by a previous thread. Avoid using any mechanism other than `$SETAST` to block ASTs; it will interfere with the process as a whole and may produce undesirable results.

- Another implication is that it is possible for a thread to be executing on one virtual processor at the same time that an AST is executing on another virtual processor. In general, this should not pose a significant problem for multithreaded applications. Such applications should have already minimized their AST use, since ASTs and threads can be difficult to use together reliably. In addition, AST routines should already be performing only atomic operations, since thread synchronization is not available to code executing at AST level. Any "legacy" code (such as a nonthreaded application using threaded libraries) is executed in the initial thread, where the normal assumptions about AST delivery are maintained. If a piece of code cannot tolerate concurrent execution with an AST routine, it should disable AST delivery during its execution.

### B.12.3 Blocking System Services

All blocking system services are thread synchronous in OpenVMS Alpha Version 7.0. That is, they block only the calling thread. When the thread is to be blocked by the system service, the OpenVMS scheduler makes an upcall to allow DECthreads to schedule another user thread to execute. Therefore, only the calling thread is blocked. All other threads are unaffected, and the process continues running. When the service completes, the thread is awakened by means of another upcall, and DECthreads schedules it to run again at the thread's next opportunity.

This applies to all "W" forms of system services. For example, `$QIOW`, `$END_TRANSW`, and `$GETJPIW`. Additionally, this applies to the following event flag services: `$WAITFR`, `$WFLAND`, and `$WFLOP`.

### B.12.4 \$HIBER and \$WAKE

`$HIBER` and `$WAKE` result in upcalls to DECthreads. When a user thread calls `$HIBER`, only that thread is blocked; all other threads continue running. The blocking thread is immediately unscheduled and another thread is scheduled to run instead. When a thread (or another process) calls `$WAKE`, all hibernating threads are awakened.

Prior to OpenVMS Version 7.0, a thread which called a `$HIBER` (or called a library routine which eventually resulted in a call to `$HIBER`) would cause the whole process to hibernate for a brief period whenever that thread was scheduled to "run". Also, with multiple threads in calls to `$HIBER` simultaneously, there was no reliable way to wake the threads (or a specific thread); the next hibernating thread to be scheduled would awaken, and any other threads would continue to sleep.

In OpenVMS Alpha Version 7.0, these problems are resolved. However, this new behavior has some other effects. For instance, hibernation-based services, such as `LIB$WAIT` and the C RTL `sleep()` routine, may be prone to premature completion. If the service does not validate its wakeup (that is, ensure that enough time has passed or that there is some other reason for it to return), then it will be prone to this problem, as are the above services, since they do not perform such wake-up validation.



### B.12.5 Event Flags

All event flags are shared by all threads in the process. Therefore, it is possible for different threads' use of the same event flag to cause interference. That is, if two threads use the same event flag in calls to different system services, whichever service completes first will cause both threads to awaken, even though the other service has not completed. This situation can be resolved by specifying an I/O Status Block (IOSB) for those system services that use them. When an IOSB is present, the blocked thread will not be awakened when the event flag is set, unless the IOSB has also been written.

Note that a DECthreads process will rarely be in "LEF" state. In general, instead of blocking for an event flag wait, DECthreads will schedule another thread to be run. If no threads are available, DECthreads will schedule a "null" thread which will place the virtual processor in "HIB" state until it is needed to execute a thread.

Note that no upcall is made for waits on a common event flag. If a thread waits on a common event flag, the virtual processor will block until the wait is satisfied. (On a uniprocessor, this will most likely block all threads.)

### B.12.6 Interactions with OpenVMS

There are several interactions with the OpenVMS operating system that should be noted:

- Like system service calls, pagefault waits are also thread-synchronous. When a thread incurs a "hard" pagefault (one which involves reading the page from disk), an upcall is performed and the thread is placed in a blocked state. DECthreads schedules another thread to run in its place. When the pagefault resolution is complete, another upcall occurs and DECthreads schedules it to run at its next opportunity. It is possible for multiple threads to take faults on the same page at approximately the same time. Each of them is blocked in turn, and they are all unblocked when the page becomes valid.
- Most OpenVMS system services would not be able to tolerate being called by multiple threads concurrently. Therefore, calls to OpenVMS system services are serialized, by means of a mechanism called the **inner mode semaphore**. If a second thread attempts to call a system service while another thread is in the middle of calling a system service, the second thread will be blocked by an upcall until the first thread completes its service call.
- DECthreads timeslicing has changed slightly under OpenVMS Alpha Version 7.0. Prior to Version 7.0, the DECthreads timeslicer was implemented using an OpenVMS timer. This caused a DECthreads scheduler AST to be delivered to the process at regular wall-clock time intervals. While running on wall-clock time was a necessary evil (in order to be able to interrupt system service blocks), this timeslice mechanism had several drawbacks. In Version 7.0, timeslicing is implemented by means of an upcall that is delivered after the thread has consumed a sufficient amount of CPU time. Thus, when no threads are running, no timeslicing takes place.



## B.12.7 Image Exit

In processes that use threading, image exit occurs as follows: `$EXIT` does not immediately invoke exit handler routines. Instead, it results in an upcall which causes DECthreads to schedule a special thread to execute the exit handler routines. `$EXIT` then calls `pthread_exit()` to terminate the calling thread. This allows it to release any resources that it might be holding.

The exit handler routines are executed in a separate thread to avoid entering a deadlock situation. For example, if the thread that called `$EXIT` did so while holding a mutex which is required by an exit handler routine, then the exit handler routine would cause the thread to block forever, waiting for a mutex that it already holds. Having the exit handler executed in a separate thread allows that thread to block while the thread holding the mutex cleans up.

`$FORCEX` works in an analogous fashion. Instead of invoking `$EXIT` directly, it causes an upcall that allows DECthreads to release the exit-handler thread.

DCL Control-Y continues to work as it always has on multithreaded applications. However, typing `EXIT` or issuing any other command that invokes a new image causes the `$FORCEX` upcall. While this is an improvement over the pre-Version 7.0 behavior in many cases, it does not guarantee that the multithreaded application will exit. For example, if the application is deadlocked, holding a resource required by one of the exit handler's routines, the application will continue to hang - even after typing Control-Y and `EXIT`. In these cases, type Control-Y again and issue the `STOP` command. This will terminate the application without running exit handlers. Note that the application will be unable to clean up, and it may leave data files and the terminal in an inconsistent state.

## B.12.8 Sysgen Parameter

The `SYSGEN` parameter `MULTITHREAD` limits the maximum number of kernel threads per process. It is set by `AUTOGEN` to the number of CPUs on the system. If `MULTITHREAD` is set to zero (0), two-level scheduling support is disabled, and DECthreads reverts to pre-Version 7.0 behavior: no upcalls will occur, and DECthreads will not utilize all processors in multiprocessor systems.

## B.12.9 Process Control System Services and DCL Commands

### B.12.9.1 Process-Level System Services

The following system services continue to be process-based: `$SUSPEND`, `$RESUME`, and `$DELPRC`. These services will operate on an entire process; they are not thread-based. For example, `$SUSPEND` issued by a thread will suspend all of the virtual processors in process, not just the calling thread. Note that on Version 7.0 you may see all but one of your kernel threads in `SUSP` state at certain times (such as when at a breakpoint in the debugger). This effect is a part of the debugging support and is not the result of calling `$SUSPND`.

### B.12.9.2 Kernel-Level System Services

The following system services now operate on a per-thread basis: `$HIBER`, `$SCHDWK`, and `$SYNCH`. These services will not operate on an entire process; they are thread-based. For example, `$HIBER` will cause the calling thread to become inactive, but will not affect other threads in the process.



### **B.12.9.3 DCL Commands**

The following DCL commands operate as indicated:

- **STOP/IDENTIFICATION** - This command continues to work on a process basis.
- **SET PROCESS** - This command continues to work on a process basis, except for **SET PROCESS/PRIORITY**.
- **SET PROCESS/PRIORITY** - This command now sets the priority of a kernel thread. Avoid setting different priorities for kernel threads in the same process. Refer to Section B.12.1.







---

## Considerations for Windows NT and Windows 95 Systems

This appendix discusses DECthreads issues and restrictions specific to the Windows NT and Windows 95 operating systems.

### C.1 Using DECthreads Routines on Windows NT and Windows 95 Systems

DECthreads is supported in the Win32 subsystem of the Windows NT operating system and Windows 95. The Win32 subsystem provides support for multithreading through the Win32 Application Programmer's Interface (API). The Win32 API allows for thread creation, termination, synchronization, and other thread functions. DECthreads adds value to the Win32 API by providing the POSIX 1003.4a API as well as the other DECthreads APIs, which are also available across all Digital platforms and other DCE platforms. The POSIX 1003.4a API allows for ease of portability of applications written to it.

---

#### Note

Note that the final POSIX 1003.1c System Application Program Interface and the tis interface are not yet available for Windows NT and Windows 95 systems as of this manual's publication date. Only the CMA and POSIX 1003.4a Draft 4 appendices apply to application development on Windows NT and Windows 95 at this time.

---

The DECthreads APIs and the Win32 threads API are interoperable. Threads created using the Win32 API can use DECthreads synchronization primitives, and threads created using one of the DECthreads APIs can use the Win32 synchronization primitives. See Section C.5 for additional information.

### C.2 Compiling DECthreads Applications

Compile all multithreaded applications on Windows NT and Windows 95 systems with the `-D_MT` switch. This ensures that reentrant definitions, such as `errno`, are used with the application being built. Applications should also use the `-DWIN32` and `-D_DLL` compiler switches. For example:

```
% CL /c myprog.c -D_MT -DWIN32 -D_DLL
```

The appropriate switches for compiling a multithreaded application can be obtained by using the `$(CVARSMTDLL)` switch found in the `NTWIN32.MAK` file. See the *Building Apps/DLLs QuickRef* documentation available through the *Win32 SDK Online References (Common)* online help icon for more build details.



### C.3 Linking DECthreads Applications

Applications using C Run-Time Library functions should be linked against one of the C Run-Time Libraries that support multithreaded applications—either `LIBCMT.LIB`, `CRTDLL.LIB` or `MSVCRT.LIB` (depending on the compiler version being used.) For example:

```
% CL /link myprog.obj CRTDLL.LIB
```

`LIBCMT.LIB` is a statically linked library that supports multithreading. `CRTDLL.LIB` is an export library for `CRTDLL.DLL`, a dynamically linked library that also supports multithreading. Multithreaded applications should not link against `LIBC.LIB` because it does not support multithreaded applications.

### C.4 File Naming Support for DECthreads Header Files

DECthreads provides the following public header files:

- `cma.h`
- `exc_handling.h`
- `pthread.h`
- `pthread_exc.h`

Both `exc_handling.h` and `pthread_exc.h` are public files, and they may be included in customer code.

### C.5 Win32/DECthreads API Interoperability

Threads created by DECthreads are fully interoperable with the Win32 API and may create and use Win32 synchronization elements and other constructs. Threads created using the Win32 API routine, `CreateThread()`, may also create and use DECthreads (POSIX 1003.4a) synchronization elements and other constructs.

The first time a thread created using the Win32 API makes a call into the DECthreads API, it becomes registered with the DECthreads package and is allowed to make use of DECthreads primitives. This type of thread should not call `pthread_exit()` because Win32 API threads are not set up to exit like a DECthreads API thread.

DECthreads considers threads created by the Win32 API detached by default and does not allow any other thread to join it. However, any Win32 created thread can join a DECthreads API created thread.

Threads created using the C Run-Time Library function, `_beginthread()`, will interoperate with DECthreads the same way as a thread created using the Win32 API routine, `CreateThread()`.

DECthreads structured exception handling is interoperable with Microsoft C exception handling, but has slightly different syntax. DECthreads supports a different set of tokens for establishing exception blocks. The DECthreads tokens include `TRY`, `CATCH`, `CATCH_ALL`, `FINALLY` and `ENTRY`. The Microsoft tokens include `try` and `except`. The DECthreads exception semantics support multiple catch clauses and provide support for creating unique (address)



exceptions. For more information about using the DECthreads exception-handling package, see Chapter 5. See your Microsoft C documentation for Microsoft C exception-handling information.

## C.6 Restrictions for Win32 API Routines

It is strongly recommended that the Win32 API routine `TerminateThread()` not be used. This routine terminates a thread immediately, but does not provide a means for the thread to clean up any context it may have acquired while executing. Context that is left in an inconsistent state when calling `TerminateThread()`, such as a locked mutex, can cause unpredictable results. The only time that this routine should be used is if the entire process is being terminated one thread at a time. Terminating just one thread in an application with this routine, and allowing the process to continue, is not recommended.

The Win32 API routine `ExitThread()` should not be called by a thread created using any of the DECthreads creation routines (for example, `pthread_create()`). `ExitThread()` bypasses normal cleanup handlers and internal management routines. `ExitThread()` also leaves internal data structures in an inconsistent state and may produce unpredictable results.

## C.7 Thread Cancelability of System Services

System calls are not cancellation points. None of the system calls should be called with asynchronous cancellation enabled. For more information, see Section 2.6.

## C.8 Unsupported DECthreads Interface Routines

The following DECthreads interface routines are not implemented and not supported on Windows NT or Windows95:

- `cma_attr_set_sched/cma_attr_get_sched`
- `cma_attr_set_priority/cma_attr_get_priority`
- `cma_thread_set_sched/cma_thread_get_sched`
- `cma_thread_set_priority/cma_thread_get_priority`
- `pthread_attr_setsched/pthread_attr_getsched`
- `pthread_setscheduler/pthread_getscheduler`
- `pthread_attr_setprio/pthread_attr_getprio`
- `pthread_setprio/pthread_getprio`
- `pthread_attr_getguardsize_np/pthread_attr_setguardsize_np`

If these routines are called by your application (because, for example, you are using the same code base on multiple Digital operating systems), and you do not wish to remove them, then ensure that your code checks these functions' return values for unimplemented/unsupported or catches the unimplemented exception and takes appropriate action. The alternative is to conditionally compile the unsupported routine calls above.

Calling one of the above `pthread_*` routines will return a -1 and set `errno` to `ENOSYS`.

Calling one of the above `cma_*` routines will raise the `cma_e_unimp` exception.



The following DECthreads interface routines **are not implemented**, and do not return errors or generate exceptions. **Do not use these routines.** They will be fixed in a future release to return errors and generate exceptions.

- `cma_attr_get_guardsize/cma_attr_set_guardsize`
- `cma_attr_set_inherit_sched/cma_attr_get_inherit_sched`
- `pthread_attr_setinheritsched/pthread_attr_getinheritsched`
- `cma_stack_check_limit_np`

The `cma_attr_set_stacksize` and `pthread_attr_setstacksize` interface routines do not change the stacksize of newly created threads. DECthreads threads on NT are created with their stack size set to the same size as the primary thread of the process they are created in. The stack size grows as needed. See the Win32 documentation for `CreateThread()`. No errors or exceptions are generated as a result of using the `cma_attr_set_stacksize` and `pthread_attr_setstacksize` routines. Their corresponding routines `cma_attr_get_stacksize` and `pthread_attr_getstacksize` return the values set using `cma_attr_set_stacksize` and `pthread_attr_setstacksize`; but this is not useful since DECthreads threads are not created using the stack size attribute.

## C.9 Use Restrictions of `cma_delay` and `pthread_delay_np` Routines

These routines accept a maximum `time_interval/interval` value of 4294967.295 seconds. Values greater than this will raise the `cma_e_badparam` exception for `cma_delay` and return -1 and set `errno` to `EINVAL` for `pthread_delay_np`.

## C.10 Timing Issues

During calls to `cma_delay`, `pthread_delay_np`, `pthdexc_delay_np`, `cma_timed_cond_wait`, `pthread_cond_timedwait`, and `pthdexc_cond_timedwait`, an application might return before the expiration of the entire time interval specified in the call with no evidence of a premature return (exception or return status). This early return is a problem with the internal Win32 behavior of `WaitForSingleObject()`. The return might be premature up to 0.05 seconds of what the desired delay was set for.



---

## Debugging Multithreaded Applications

The debugging information in this appendix is specific to applications that utilize DECthreads. It describes how to use the DECthreads Debugger and provides additional debugging information for the Windows NT programmer.

### D.1 DECthreads Debugger

The DECthreads Debugger is a powerful tool designed to complement your existing debugger. It provides you with thread-specific command-string debugging, statistical and historical thread information on request, and many useful thread debugging commands and qualifiers. To invoke the DECthreads Debugger:

- From DECladebug on Digital UNIX systems, enter:

```
print pthread_debug() or call pthread_debug()
```

- From the OpenVMS debugger, enter:

```
SET IMAGE PTHREAD$RTL  
CALL PTHREAD_DEBUG
```

Note that PTHREAD\_DEBUG must be entered in capital letters if the language is set to C or C++.

You may wish to define a DEBUG command symbol, such as the following:

```
define/command pthread_debug= "set image pthread$rtl; call PTHREAD_DEBUG"
```

- A client program may call the function pthread\_debug() at any point during its execution.

---

#### Notes

---

The DECthreads Debugger must be run within the context of a program being debugged. You cannot apply it to a core dump.

The syntax of commands (and the list of supported commands) will change over time according to the needs of individual platforms. Never use calls to pthread\_debug or pthread\_debug\_cmd as a part of a production application. It is for debugging use only, and may be phased out when system debuggers take on all required debugging capabilities.

---



### D.1.1 Interactive Debugging

The `pthread_debug` subsystem prompts for commands until the `exit` command is entered (Ctrl/Z on OpenVMS systems, or Ctrl/D on UNIX systems). Entering an unrecognized command results in a short help message listing the valid commands and arguments. The `help` command gives more extensive information on the switches available for each command.

The command syntax resembles UNIX shell syntax: a command name followed by a list of arguments and switches, separated by blanks (spaces and tabs). The arguments and switches may appear in any order. The command name itself may be abbreviated. You can combine multiple switches into a single argument. For example:

```
thread -tf instead of thread -t -f.
```

### D.1.2 Noninteractive Debugging

Instead of calling `pthread_debug`, you can call `pthread_debug_cmd`. The `pthread_debug_cmd` function implements the same commands, except that instead of prompting it processes command strings specified as an argument to the function. You can include multiple commands by separating them with semicolons (;). For example:

```
pthread_debug_cmd ("thread -if;cond -f");
```

The `pthread_debug_cmd` function returns a value representing the status of the last command executed. When called from a program, you can check the return status values listed in Table D-1 against the various status constants defined in the `pthread.h` header file.

**Table D-1 DECthreads Debugging Return Status**

Return Status	Meaning
PTHREAD_DBG_SUCCESS	Command was successful.
PTHREAD_DBG_QUIT	Last command was quit or exit.
PTHREAD_DBG_NONESEL	No objects were selected (no output).
PTHREAD_DBG_NOPRIV	Unable to complete command due to privilege violation.
PTHREAD_DBG_INVPARAM	Invalid command parameter (such as bad policy keyword).
PTHREAD_DBG_INVSEQ	Invalid sequence number.
PTHREAD_DBG_INCONSTATE	Inconsistent state detected in internal data; this may mean a thread is currently modifying data.
PTHREAD_DBG_CORRUPT	Corrupt data detected (inconsistent state prevents the command from continuing).
PTHREAD_DBG_INVOPTION	Invalid command option specified.
PTHREAD_DBG_NOARG	No arguments specified to command that requires them.
PTHREAD_DBG_INVADDR	Invalid address argument specified.
PTHREAD_DBG_INVCMD	Invalid command specified.
PTHREAD_DBG_NULLCMD	No command specified.



### D.1.3 Running DECthreads in Metered Mode

Metering tells DECthreads to collect statistical and historical information on the use of synchronization objects within your program. This affects all synchronization within the program, including that within DECthreads itself and any other libraries that may use threads. Therefore, metering provides a very powerful tool for debugging multithreaded code.

To enable metering, define the environment variable (or OpenVMS logical name) `PTHREAD_CONFIG` prior to running any threaded application. The variable (or logical name) should have a value of "meter=all" to enable metering. This causes DECthreads to gather and record statistics and history information for all synchronization operations. Additionally, when running in metered mode DECthreads "poisons" all thread stacks (except the main thread stack) with a specific pattern. When using the `thread -f` or `stack` command in `pthread_debug`, DECthreads computes the number of bytes actually modified by the program on each stack.

Programs running in metered mode are somewhat slower than unmetered programs. Also, metered "normal" mutexes behave like "errorcheck" mutexes in many ways. This does not affect the behavior of correct programs, but there are some differences between normal and errorcheck mutexes of which you need to be aware. The most important difference is that normal mutexes do not report a number of usage errors, while errorcheck mutexes do.

The following operations are illegal and are always reported as errors when used on errorcheck mutexes, but may not be reported for non-metered "normal" mutexes:

- Locking a mutex in one thread and then unlocking it in another thread
- Unlocking a mutex that is not currently locked
- Relocking a mutex that the thread already owns

Since it can be expensive to detect these conditions, a normal mutex may not always report these errors. Regardless of whether the program seems to work correctly under these circumstances, the operations are illegal. A metered "normal" mutex will report these errors under more circumstances than a normal mutex that is not metered.

### D.1.4 DECthreads Debugger Commands

DECthreads Debugger commands and their qualifiers are listed in the following sections. Individual DECthreads objects may be referenced by their sequence number. Each object has a number that is unique within its object class; this number is displayed when an object is listed.

#### D.1.4.1 Broadcast Command

The broadcast command wakes all threads waiting on a condition variable. The thread does not actually awaken until after the `pthread_debug` command returns to the program, but it will happen eventually. You can use this command freely, because an unexpected broadcast is effectively a "spurious wakeup." Any code using condition variable waits must be able to handle spurious wakes correctly.

**Format:**

`broadcast [condition-variable-number]`

**condition-variable-number**



The sequence number of a condition variable to broadcast.

#### D.1.4.2 Conditions Command

The conditions command lists all known condition variables in use.

DECthreads may not know about condition variables that were statically initialized and have not been used recently. It cannot keep track of each pthread\_cond\_t structure within a program - only "auxiliary" structures that are allocated within DECthreads when a thread blocks on a condition variable. When metering, all condition variables that have ever been used are known.

##### Format:

conditions [condition-variable-number] [qualifier]

##### condition-variable-number

One or more object sequence numbers separated by spaces. If specified, only those condition variables whose sequence numbers are listed are displayed. Each argument that does not begin with "-" is interpreted as the sequence number of a condition variable.

Qualifier	Description
-a	Include internal condition variables created by DECthreads.
-f	Display complete information about the state of each condition variable, including whether there are deferred signal operations or pending wakeups that need to be processed.
-h	Display the history buffer of recent operations carried out on each condition variable. This history is maintained only when DECthreads is run with metering enabled. Each line in the history shows a specific operation (wait, signal, or broadcast) with additional information such as whether a signal or broadcast awakened another thread. It also shows the thread that invoked the function and the program PC from which DECthreads was called. This information is available when DECthreads is run with metering enabled.
-q	Display the sequence number of each thread that is waiting on a condition variable (if there are any waiters).
-s	Display statistics for each condition variable. This includes the number of waits, signals, and broadcasts, the number of times a thread actually blocked on the condition variable, the maximum and average number of threads blocked on the condition variable at any time. This information is available when DECthreads is run with metering enabled.
-w	List only condition variables that have threads waiting on them.

#### D.1.4.3 Exit and Quit Commands

Exit or quit pthread\_debug or pthread\_debug\_cmd and return to the program (or the debugger) immediately.

##### Format:

exit or quit



#### D.1.4.4 Help Command

The help command describes the commands and options available within pthread\_debug. Without arguments, help will provide a brief one-line description of each command, with a list of the options supported. You can also specify an argument; help will give a full description of all commands having the argument as an initial string.

**Format:**

help [command]

**command**

Any DECthreads Debugger command. For example, help t will describe the thread and tset commands in detail. To get a full description of all commands, you can type help \*. When you enter a command that is not recognized, pthread\_debug will display the same output as help without arguments.

#### D.1.4.5 Keys Command

The keys command displays the thread-specific data keys currently defined in the process. They are identified by sequential integers. The address of the destructor routine (if any) is displayed for each.

**Format:**

keys

#### D.1.4.6 Mutexes Command

List all known mutexes in use. DECthreads may not know about mutexes that were statically initialized and haven't been used recently. It cannot keep track of each pthread\_mutex\_t structure within a program - only "auxiliary" structures that are allocated within DECthreads when a thread blocks on a mutex. When metering, all mutexes that have ever been used are known.

**Format:**

mutexes [object-number] [qualifier]

**object-number**

One or more object sequence numbers separated by spaces. If specified, only those objects whose sequence numbers are listed are displayed. Each argument that does not begin with "-" is interpreted as the sequence number of an object.

Qualifier	Description
-a	Include internal mutexes created by DECthreads.
-f	List additional information for each mutex.
-h	Display the history buffer of recent operations carried out on each mutex. This history is maintained only when DECthreads is run with metering enabled. Each line in the history shows a specific operation (lock or unlock) with additional information such as whether an unlock awakened another thread and whether a lock attempt resulted in blocking the thread. It also shows the thread that invoked the function and the program PC from which DECthreads was called. This information is available when DECthreads is run with metering enabled.
-l	List only locked mutexes.



Qualifier	Description
-q	Display the sequence number of each thread waiting to lock a mutex (if there are any waiters).
-s	Display statistics for each mutex. This includes the number of locks and unlocks, the number of times a thread actually blocked on the mutex, the maximum number of threads blocked on the mutex at any time, and an average "contention" for the mutex, which is the average number of waiters. This information is available when DECthreads is run with metering enabled.

#### D.1.4.7 Signal Command

The `signal` command wakes all threads waiting on a condition variable. The thread does not actually awaken until after the `pthread_debug` command returns to the program, but it will happen eventually. You can use this command freely, because an unexpected broadcast is effectively a "spurious wakeup". Any code using condition variable waits must be able to handle spurious wakes correctly.

##### Format:

```
signal [condition-variable-number]
```

##### condition-variable-number

The sequence number of a condition variable to signal.

#### D.1.4.8 Show Command

The `show` command displays various DECthreads information.

##### Format:

```
show [qualifier]
```

Qualifier	Description
-c	Identify the thread currently executing the <code>show -c</code> command.
-s	List the scheduling policies supported by DECthreads, with the range of priorities allowed for each.
-v	Display the state of all current kernel "threads" in the process, whether active or cached (valid in a DECthreads implementation that uses kernel-supplied concurrency objects).

#### D.1.4.9 Stack Command

The `stack` command displays information on DECthreads stacks. It displays the start and end address of each stack area. When running in metered mode, it also displays an estimate of the number of bytes used on each thread's stack (except for the default stack).

##### Format:

```
stack [stack-number] [qualifier]
```

##### stack-number

Each argument that does not begin with "-" is interpreted as an address within some thread's stack. This will cause the command to give information only on the stacks containing the specified addresses. An address is specified in C language syntax as follows: <NUMBER> for decimal, 0<NUMBER> for octal, or 0x<NUMBER> for hexadecimal.)



Qualifier	Description
-f	List the start and end address of the stack guard pages for each allocated stack.
-s	Include statistics for each stack. Currently, the only statistic reported is the maximum number of bytes written by the thread. This information can only be calculated for stacks other than the default process stack. This information is available when DECthreads is run with metering enabled.

#### D.1.4.10 System Command

The `system` command displays information about the system on which you are running. This includes the name of the hardware (where DECthreads can determine it), for example "DEC 3000 model 300 AXP", the size of physical memory, and the number of CPUs.

##### Format:

`system`

#### D.1.4.11 Threads Command

The `threads` command lists all known threads except internal threads created by DECthreads (including the null threads, which are run when a virtual processor can find no other thread to execute).

##### Format:

`threads [threads-number] [qualifier]`

##### threads-number

One or more sequence numbers separated by spaces. If specified, only those objects whose sequence numbers are listed are displayed. Each argument that does not begin with "-" is interpreted as the sequence number of an object.

Qualifier	Description
-l	(Digit "1") Display threads in a one-line tabular format.
-a	Include internal threads created by DECthreads.
-b	List threads that are currently blocked (waiting on a condition variable or attempting to lock a mutex).
-c	List the threads that are currently running. (On a uniprocessor implementation, there is only one at any time.)
-d	List the threads that have been detached. ( <code>cma_detach</code> or <code>pthread_detach</code> called for the thread, or it was created with the detached attribute set.)
-f	Give full information on threads. Essentially, this sets all the "display" selection flags. Always use -f and -a qualifiers when submitting debug output as a part of a problem report.
-h	List the threads that have been held using the <code>tset -h</code> command.
-k	Display the thread-specific data values associated with each selected thread.
-l	List threads that own mutexes. (Normal mutexes are not included unless metering is enabled.)
-m	Display only the current thread.
-n	List the threads that have not been held.



Qualifier	Description
-o	Select specific output fields. For example, thread -o res. b : blocking information c : cancel status i : "internal" state k : tsd keys l : show owned mutexes (if known) m : miscellaneous state r : realtime scheduling information s : stack information v : virtual processor state
-r	List the threads that are ready to run. (They are not blocked or terminated, but cannot currently run because the processor is busy.)
-s <i>policy</i> [ <i>operator priority</i> ]	List threads with particular scheduling policies and priorities. The policy field is required and specifies the name of a scheduling policy: <i>fifo</i> , <i>rr</i> , <i>foreground</i> , <i>ada_rtb</i> , <i>ada_low</i> , or <i>idle</i> . The operator field specifies a C relational operator that is to be applied in comparing a thread's priority against the -s criteria: <i>=</i> , <i>!=</i> , <i>&gt;</i> , <i>&lt;</i> , <i>&gt;=</i> , <i>&lt;=</i> . Finally, the priority field is a priority value within the priority range for the specified scheduling policy; the keywords <i>min</i> (minimum priority), <i>mid</i> (midrange priority), and <i>max</i> (maximum priority) are accepted as well as integer values.  The operator and priority fields are optional. If both are omitted, all threads with the specified scheduling policy will be selected.
-t	List the threads that have terminated, but have not yet been detached. (They are waiting for another thread to join with them.)

#### D.1.4.12 Tset Command

The **tset** command alters the state of the specified threads.

##### Format:

**tset** threads-number [qualifier]

##### threads-number

Use one or more thread numbers separated by spaces, or use the asterisk character (\*) to alter all threads. Do not use the asterisk character (\*) with the -c qualifier.

Qualifier	Description
-c	Cancel (or, in CMA interface terms, alert) the specified thread. The cancellation will take effect after the thread resumes execution.
-h	Set one or all threads to "hold" state. Threads "on hold" will not be executed until they are "unheld" using <b>tset -n</b> .
-n	"Unhold" one or all threads that are currently held. They will be eligible to run in the future as if they had not been held, but will not necessarily run immediately (they will never run until you exit <b>pthread_debug</b> ).



Qualifier	Description
-s [policy]:priority	<p>Set the scheduling policy and priority of one or all threads. The value of the switch is the next argument on the command line, and cannot contain spaces. The <i>policy</i> field is optional and specifies the name of a scheduling policy: <i>fifo</i>, <i>rr</i>, <i>foreground</i>, <i>background</i>, <i>ada_rtb</i>, <i>ada_low</i>, or <i>idle</i>. The <i>priority</i> field is a priority value within the priority range for the specified scheduling policy; the keywords <i>min</i> (minimum priority), <i>mid</i> (midrange priority), and <i>max</i> (maximum priority) are accepted as well as integer values.</p> <p>When you omit the <i>policy</i> argument, only the priority of the thread changes. In this case, the priority keywords are interpreted relative to the current scheduling policy of the thread, and an integer priority value must be within that policy's range.</p>

#### D.1.4.13 Versions Command

The `versions` command displays the current version of DECthreads, the type of operating system and hardware for which it was built, and the current operating system version.

##### Format:

`versions`

#### D.1.4.14 VM Command

The `vm` command displays information on the DECthreads internal memory management system. DECthreads allocates process memory using `malloc` on UNIX operating systems, and either `lib$vm_malloc` or `lib$get_vm_page` on OpenVMS operating systems (the latter for stacks, and the former for all other memory requirements).

Memory is cached internally, for performance, at several different levels. "Live" objects that contain a state that is relatively expensive to recreate are cached intact for efficient reuse. Additionally, "raw" memory of common sizes is cached for general reuse.

##### Format:

`vm [qualifier]`

Qualifier	Description
-c	List the intact object cache as well as the "raw memory" cache.
-f	List all of the statistics maintained for each cache, including the adaptive highwater marking information. Each cache list is controlled by a highwater mark specifying the maximum number of objects that can be contained. When a cache reaches the maximum, objects are returned to the external memory manager to avoid starving the rest of the process. DECthreads adjusts the high water mark for each cache based on estimates of the average usage, and the memory command displays all of the statistics that control this adjustment.



## D.2 Debugging Threads on Windows NT Systems

The WinDbg Debugger, provided with the Windows NT Software Development Kit (SDK), provides support for debugging a threaded program. When working with DECthreads in WinDbg, be aware that terminating a thread with `pthread_exit()`, `cma_exit_normal()`, or `cma_exit_error()` raises an exception as a normal part of the exiting process. Unless you explicitly tell WinDbg to ignore this exception, it catches the exception and stops execution. To allow the thread to exit properly and your application to continue, use the WinDbg `gn` (go not handled) command. This command tells WinDbg to ignore the exception.

WinDbg also stops at each DECthreads exception block for this exception and has to be continued using the WinDbg `gn` command until the thread's stack is completely unwound. The exception used by DECthreads for thread rundown is `0xC1380034`.

DECthreads also generates an exception in a thread as a normal part of thread cancellation. When cancelling a thread with `pthread_cancel()` or `cma_thread_alert()`, an exception value of `0xC1380034` is raised. This exception should be allowed to propagate all the way to the base of the thread's stack. If WinDbg halts execution of the program and indicates that this exception is being raised, use the WinDbg `gn` command to allow the exception to continue.



---

## Digital Proprietary Interface Routines: CMA

This appendix provides detailed descriptions of the (**cma**) routines that are part of the Digital proprietary interfaces to DECthreads.

---

### Note

In future releases, the CMA interface used with DECthreads will continue to exist and be supported in the OpenVMS operating system, but it will no longer be documented or enhanced. Therefore, Digital recommends that you port your CMA-based application to the IEEE Std 1003.1c-1995, POSIX System Application Program Interface provided by DECthreads. In addition to providing CMA routine documentation, this appendix also gives hints about migrating your application from using the CMA interface to using the new POSIX 1003.1c pthread interface.

---

It may be beneficial to save the previous version of this guide if you intend to support an application written using CMA for many years.

To indicate errors, the CMA routines raise exceptions. See Section E.4 for exception descriptions.

## E.1 Migrating from a CMA Interface to POSIX 1003.1c

The new POSIX 1003.1c DECthreads interface is significantly different than the CMA interface, though there are many similarities between the functions individual routines perform. This section gives hints about the relationship between the two sets of routines to assist in migration of applications from using the CMA interface to the DECthreads POSIX 1003.1c interface. For further routine information, refer to Part II of this guide, which describes each POSIX 1003.1c Interface routine.

Note that routine names ending with the `_np` suffix denote that the routine is not portable—the routine might not be available in implementations of POSIX 1003.1c other than DECthreads. You need not prototype the pthread routines if you include `pthread.h`.

### E.1.1 CMA Handles

A **cma handle** is storage, similar to a pointer, that refers to a specific DECthreads object (thread, mutex, condition variable, queue, or attributes object).

Handles are allocated by the user application. They can be freely copied by the program and stored in any class of storage; objects are managed by DECthreads.



Because DECthreads objects are only accessed by handles in the cma interface, you can think of the handle as if it were the object itself.

DECthreads objects are accessed by handles in the cma interface, rather than pointers, because handles allow for greater robustness and portability. Handles allow DECthreads to detect the following types of run-time errors:

- Using an uninitialized handle
- Using a handle that has been corrupted
- Using a handle whose object no longer exists (a dangling handle)

Handles do not exist in the POSIX 1003.1c standard interface. While providing less robustness due to more limited error checking, this allows better performance by decreasing memory use and memory access (handles result in pointers to pointers).

### E.1.2 Interface Routine Mapping

Many CMA routines perform functions nearly identical to the POSIX 1003.1c interface routines. The syntax and semantics may differ, but there is nonetheless quite a bit of similarity.

CMA Routine	POSIX 1003.1c Routine	Notes
cma_alert_disable_async	pthread_setcancelstate/pthread_setcanceltype	
cma_alert_disable_general	pthread_setcancelstate/pthread_setcanceltype	
cma_alert_enable_async	pthread_setcancelstate/pthread_setcanceltype	
cma_alert_enable_general	pthread_setcancelstate/pthread_setcanceltype	
cma_alert_restore	pthread_setcancelstate/pthread_setcanceltype	
cma_alert_test	pthread_testcancel	
cma_attr_create	pthread_attr_init	
cma_attr_delete	pthread_attr_destroy	
cma_attr_get_guardsize	pthread_attr_getguardsize_np	
cma_attr_get_inherit_sched	pthread_attr_getinheritsched	
cma_attr_get_mutex_kind	pthread_mutexattr_gettype_np	
cma_attr_get_priority	pthread_attr_setsched_param	
cma_attr_get_sched	pthread_attr_getschedpolicy	
cma_attr_get_stacksize	pthread_attr_getstacksize	
cma_attr_set_guardsize	pthread_attr_setguardsize_np	
cma_attr_set_inherit_sched	pthread_attr_setinheritsched	
cma_attr_set_mutex_kind	pthread_mutexattr_settype_np	
cma_attr_set_priority	pthread_attr_setsched_param	
cma_attr_set_sched	pthread_attr_setschedpolicy	
cma_attr_set_stacksize	pthread_attr_setstacksize	
cma_cond_broadcast	pthread_cond_broadcast	



CMA Routine	POSIX 1003.1c Routine	Notes
cma_cond_create	pthread_cond_init	
cma_cond_delete	pthread_cond_destroy	
cma_cond_signal	pthread_cond_signal	
cma_cond_signal_int	pthread_cond_signal_int_np	
cma_cond_timed_wait	pthread_cond_timedwait	
cma_cond_wait	pthread_cond_wait	
cma_debug	pthread_debug	
cma_debug_cmd	pthread_debug_cmd	
cma_delay	pthread_delay_np	
cma_handle_assign	NONE	Use Language assignment operator
cma_handle_equal	pthread_equal	
cma_init	NONE	Not necessary
cma_key_create	pthread_key_create	pthread_key_delete is available as well
cma_key_get_context	pthread_getspecific	
cma_key_set_context	pthread_setspecific	
cma_lock_global	pthread_lock_global_np	
cma_mutex_create	pthread_mutex_init	
cma_mutex_delete	pthread_mutex_delete	
cma_mutex_lock	pthread_mutex_lock	
cma_mutex_try_lock	pthread_mutex_trylock	
cma_mutex_unlock	pthread_mutex_unlock	
cma_once	pthread_once	
cma_stack_check_limit_np		
cma_thread_alert	pthread_cancel	
cma_thread_bind_to_cpu	NONE	
cma_thread_create	pthread_create	
cma_thread_detach	pthread_detach	
cma_thread_exit_error	pthread_exit	With Status
cma_thread_exit_normal	pthread_exit	With Status
cma_thread_get_priority	pthread_getschedparam	
cma_thread_get_sched	pthread_setschedparam	
cma_thread_get_self	pthread_self	
cma_thread_join	pthread_join	
cma_thread_set_priority	pthread_setschedparam	
cma_thread_set_sched	pthread_setschedparam	
cma_time_get_expiration	pthread_get_expiration_np	
cma_unlock_global	pthread_unlock_global_np	
cma_yield	pthread_yield	



### E.1.3 New POSIX 1003.1c Routines

The following are POSIX 1003.1c interface routines that do not have any functional similarities in the CMA implementation:

- pthread\_atfork (UNIX only)
- pthread\_attr\_getdetachstate
- pthread\_attr\_setdetachstate
- pthread\_key\_delete
- pthread\_kill (UNIX only)
- pthread\_sigmask (UNIX only)

## E.2 Atomic Queues

**Atomic queues** are DECthreads Library objects that you can use to communicate information among threads or among routines in a single thread.

---

#### Note

Atomic queues are documented in Appendix F, the DECthreads Library reference section. The queue routines begin with the `cma_lib_` prefix; they are not included with the pthread interface.

---

Operations on queues are atomic because any operation on the queue is guaranteed to complete before any other operation on that same queue can begin. Queue operations are not interruptable.

An atomic queue contains preallocated queue items. Each element (an integer identifier or a pointer to a block of data) inserted into the queue consumes a queue item. The number of elements allowed on the queue is called the **queue size**. The `queuesize` attribute can be specified by calling the `cma_lib_attr_set_queuesize` routine. The default is 128 queue items.

You can insert an element at the end of a queue by calling `cma_lib_queue_enqueue` or at the front of a queue by calling `cma_lib_queue_requeue`. You can remove an element from a queue by calling `cma_lib_queue_dequeue`. In each of these routines, if the element cannot be inserted or removed because the queue is full or empty, the calling thread is blocked until the action can be performed.

If you want to insert or remove an element but return with a status value if the queue is full or empty rather than cause the thread to wait, you can call the `cma_lib_queue_try_enqueue`, `cma_lib_queue_try_requeue`, or `cma_lib_queue_try_dequeue` routines. These routines return a Boolean value indicating whether or not the routine succeeded.

The `cma_lib_queue_try_enqueue_int` routine works exactly like `cma_lib_queue_try_enqueue` except that it can be called from an interrupt handler. Only routines with an `_int` suffix can be called from interrupt routines.

## E.3 Scheduling Parameters

The **scheduling parameters attribute** specifies the execution priority of a thread. The priority is expressed relative to other threads in the same policy on a continuum of minimum to maximum for each scheduling policy. A thread's priority falls within one of the following ranges, depending on its scheduling policy:



Low	Mid	High
cma_c_prio_fifo_min	cma_c_prio_fifo_mid	cma_c_prio_fifo_max
cma_c_prio_rr_min	cma_c_prio_rr_mid	cma_c_prio_rr_max
cma_c_prio_through_min	cma_c_prio_through_mid	cma_c_prio_through_max
cma_c_prio_back_min	cma_c_prio_back_mid	cma_c_prio_back_max

## E.4 CMA Exceptions

Table E-1 lists the CMA exceptions and gives an explanation of each. It also contains the recommended action you should take if an exception occurs.

**Table E-1 CMA Exceptions**

Exception	Explanation and User Action
cma_e_alerted	<p>Thread execution was alerted</p> <p><b>Explanation:</b> A thread was requested to terminate by either the <code>cma_thread_alert</code> or <code>pthread_cancel</code> routine. DECthreads uses an alert to request that a thread terminate after first performing cleanup and shutdown operations.</p> <p><b>User Action:</b> If you do not want threads to terminate at the point where this alert is being delivered, you can use several routines (<code>cma_alert_disable_general</code>, <code>cma_alert_disable_asynch</code>, <code>cma_alert_restore</code>, <code>pthread_setcancelstate</code>, and <code>pthread_setcanceltype</code>) to specify points in the thread process where alerts cannot be delivered to the thread.</p>
cma_e_alert_nesting	<p>Improper nesting of alert scope</p> <p><b>Explanation:</b> An attempt was made to restore an inner scope after an enclosing outer scope had already been restored.</p> <p><b>User Action:</b> Examine the code to determine where the incorrect alert state variable was passed to the <code>cma_alert_restore</code> routine.</p>
cma_e_badparam	<p>Parameter to DECthreads operation is invalid</p> <p><b>Explanation:</b> A parameter passed to a DECthreads routine is improper; for example, the value is of the wrong type or is out of range.</p>

(continued on next page)



**Table E-1 (Cont.) CMA Exceptions**

Exception	Explanation and User Action
	<p>User Action: Determine which routine raised the exception. Then consult the documentation to determine the correct parameters and value ranges. Update your code accordingly and retry the operation.</p>
	<p>If you continue to have problems, report it to Digital, including a small test program that reproduces the problem.</p>
<code>cma_e_existence</code>	<p>Object referenced does not currently exist</p>
	<p>Explanation: A DECthreads routine was requested to operate on an object that does not exist.</p>
	<p>User Action: Consult the documentation for the DECthreads routine that issued this message to determine the conditions that caused it. Also check the program where the call is issued to determine which object or objects that are being passed as parameters do not currently exist.</p>
<code>cma_e_exit_thread</code>	<p>Current thread was requested to exit</p>
	<p>Explanation: The <code>cma_thread_exit</code> routine was called to force the thread to shut down in an orderly fashion. This message notifies all active exception handlers to perform any necessary cleanup activities.</p>
	<p>User Action: None</p>
<code>cma_e_inialrpro</code>	<p>DECthreads initialization is already in progress</p>
	<p>Explanation: A call was made to the DECthreads initialization routine <code>cma_init</code> while DECthreads was still trying to initialize itself on a prior call. DECthreads initialization must be complete before any DECthreads routines are used. Once DECthreads is fully initialized, all calls to <code>cma_init</code> complete successfully.</p>
	<p>User Action: Remove the offending concurrent call to the <code>cma_init</code> routine or delay it until the first call to <code>cma_init</code> has completed.</p>
<code>cma_e_in_use</code>	<p>Object referenced is already in use</p>

(continued on next page)



**Table E-1 (Cont.) CMA Exceptions**

Exception	Explanation and User Action
	<p><b>Explanation:</b> The DECthreads operation cannot be performed on the specified object because it is already in use; for example, the routine is attempting to delete a mutex that is locked.</p> <p><b>User Action:</b> Determine which routine caused the error and make sure the object is in an appropriate state before attempting the operation.</p>
<code>cma_e_nostackmem</code>	<p>No space is currently available to create a new stack</p> <p><b>Explanation:</b> A call to <code>cma_thread_create</code> or another DECthreads routine requires that a new stack be created, but there is insufficient space to create it.</p> <p><b>User Action:</b> Reduce the size of thread stacks previously created, so that additional stacks may be created. Alternatively, adjust system or user quotas to allow the allocation of more virtual memory.</p>
<code>cma_e_stackovf</code>	<p>Attempted stack overflow was detected</p> <p><b>Explanation:</b> A thread overflowed its stack.</p> <p><b>User Action:</b> Recreate the thread with a larger stack or redesign the code to require less stack space; for example, nest your calls less deeply or allocate less storage on the stack.</p>
<code>cma_e_unimp</code>	<p>The specified DECthreads feature is not implemented</p> <p><b>Explanation:</b> You attempted to use a feature that is not implemented in the version of DECthreads that you are running. This error can occur when a program developed on a system running a higher version of DECthreads is executed on a system that is running a lower version of DECthreads.</p> <p><b>User Action:</b> Use a higher version of DECthreads that supports the feature or do not attempt to use the feature with a lower version of DECthreads.</p>
<code>cma_e_uninitexc</code>	<p>Uninitialized exception raised</p>

(continued on next page)



**Table E-1 (Cont.) CMA Exceptions**

Exception	Explanation and User Action
	<p>Explanation: Code using the EXC_HANDLING.H package, which provides portable exceptions for the C language, attempted to raise an exception that has not been initialized.</p> <p>User Action: Check the error messages to determine the program location where the uninitialized exception is being raised. Use the EXCEPTION_INIT macro defined in the EXC_HANDLING.H package to initialize the exception.</p>
cma_e_unkstatus	<p>Unknown exception reported</p> <p>Explanation: A status exception was raised for which DECthreads is unable to provide a meaningful text translation.</p> <p>User Action: Check the value of the status with which the exception object was initialized to make sure it has a proper text translation.</p>
cma_e_use_error	<p>Requested operation is inappropriate for the specified object</p> <p>Explanation: The state or type of an object is inappropriate for the operation; for example, the operation attempts to unlock a mutex that is not locked.</p> <p>User Action: Determine which routine caused the error and consult the documentation to learn which object states are appropriate for the routine.</p>
cma_e_wrongmutex	<p>Wrong mutex specified in condition wait</p> <p>Explanation: A thread attempted to wait for a condition variable that already has at least one thread waiting and that thread has specified a different mutex. DECthreads requires that all threads concurrently waiting for a condition variable specify the same mutex.</p> <p>User Action: Design your code so that each condition variable represents a particular state of shared data that is protected by a given mutex.</p>

See Table 5-1 for a list of pthread exceptions. Most pthread exceptions correspond directly to a CMA interface exception.



## E.5 CMA Alert Example

A thread control and condition example is shown in Example E-1.

### Example E-1 CMA Alert Example

```
/*
 * CMA Alert Example
 */

/*
 * Outermost alert scope
 */
{
    .
    .
    .
    cma_t_alert_state    outer_a_s;
    .
    .
    .
/*
 * Create a nested alert scope, saving the previous setting.  In this
 * scope, alerts will be disabled.
 */
    cma_alert_disable_general (&outer_a_s);
/*
 * Now alerts are disabled.
 */
    .
    .
    .
/*
 * Create another nested scope.  In this scope, alerts will be enabled.
 */
{
    cma_t_alert_state    inner_a_s;
    .
    .
    .
    cma_alert_enable_general (&inner_a_s);
/*
 * Now alerts are enabled.
 */
    .
    .
    .
/*
 * Create another nested scope.  In this scope, we will be enabling
 * asynchronous alerts.
 */
{
    cma_t_alert_state    async_a_s;
    .
    .
    .
/*
 * First capture the alert state, so it can be restored later.
 */
    cma_alert_disable_asynch (&async_a_s);
/*
 * Now enable asynchronous alerts.
```

(continued on next page)



### Example E-1 (Cont.) CMA Alert Example

```
    */
    cma_alert_enable_async ();
    /*
     * Now asynchronous alerts are enabled.
     */
    .
    .
    .
    /*
     * Now restore the previous alert state, which disables
     * asynchronous alerts.
     */
    cma_alert_restore (&async_a_s);
    /*
     * Now asynchronous alerts are disabled; synchronous alerts
     * are still enabled.
     */
}
.
.
.
/*
 * There is no requirement to restore all alert scopes. However, you
 * may not restore an inner scope after restoring an outer one.
 */
}
.
.
.
/*
 * Restore the original alert state.
 */
cma_alert_restore (&outer_a_s);
.
.
.
}
```

## E.6 CMA Routine Descriptions

Beginning on the next page of this appendix are descriptions of the (**cma**) routines that are a part of the Digital proprietary interfaces to DECthreads.



---

## cma\_alert\_disable\_async

Disables asynchronous alert delivery to the current thread.

### Syntax

```
cma_alert_disable_async(  
    prior);
```

Argument	Data Type	Access
prior	opaque cma_t_alert_state	write

### C Binding

```
#include <cma.h>  
  
void  
cma_alert_disable_async (  
    cma_t_alert_state *prior);
```

### Arguments

#### prior

Receives the state of asynchronous alert delivery (enabled or disabled) that exists before the call to this routine.

### Description

This routine should be used to get the current alert state immediately prior to enabling asynchronous alerts. This state can be used in a subsequent call to `cma_alert_restore` to reestablish the previous alert state. When the call to `cma_alert_disable_async` returns, call `cma_alert_enable_async` to enable asynchronous alerts. Do not call this routine to disable asynchronous alerts, rather, call `cma_alert_restore`. See Example E-1.

### Exceptions

None



## cma\_alert\_disable\_general

---

### cma\_alert\_disable\_general

Disables general delivery of alerts to the current thread.

#### Syntax

```
cma_alert_disable_general(  
    prior);
```

Argument	Data Type	Access
<i>prior</i>	opaque cma_t_alert_state	write

#### C Binding

```
#include <cma.h>  
  
void  
cma_alert_disable_general (  
    cma_t_alert_state *prior);
```

#### Arguments

**prior**

Receives the prior state of general alert delivery (enabled or disabled).

#### Description

This routine disables general delivery of alerts to the current thread and returns the previous state of alert delivery to the *prior* argument.

#### Exceptions

cma\_e\_alert\_nesting  
cma\_e\_existence  
cma\_e\_use\_error



---

## cma\_alert\_enable\_async

Enables asynchronous alert delivery to the current thread.

### Syntax

```
cma_alert_enable_async( );
```

### C Binding

```
#include <cma.h>

void
cma_alert_enable_async (void);
```

### Arguments

None

### Description

This routine enables asynchronous alert delivery to the current thread. Unlike `cma_alert_disable_async`, this routine does not return the prior state of asynchronous delivery. Before a call to this routine returns, asynchronous delivery is enabled. Any return value would be unreliable. (An alert can occur during the hardware or language procedure linkage.)

To allow restoration of the previous alert state when asynchronous cancelability is no longer needed, call `cma_alert_disable_async` to obtain the current alert state prior to calling this routine.

This routine is alertable. If an alert is pending for the current thread, and alert delivery is not currently disabled, then the pending alert is delivered before this routine returns.

Asynchronous delivery of alerts means that the `cma_e_alerted` exception can be raised at any point in the code where an interrupt can occur. The exception could potentially be raised in the middle of a hardware instruction, if that is permitted by the machine.

Asynchronous delivery of alerts is not appropriate over regions of code where resources are being allocated, or when invariants are being modified. It is difficult to determine exactly where an exception was raised within such a region. Usually this makes it very difficult (and often impossible) to properly release resources or restore invariants should an alert be delivered.

External routines often do not function correctly with asynchronous delivery of alerts enabled. Also, most existing code will not be able to cope with asynchronous alerts. Note that none of the general run-time routines are safe. None of the DECthreads library routines are safe except `cma_alert_restore` and `pthread_setasynccancel`. It is always safest to disable asynchronous delivery before calling any external routine from a region of code where asynchronous delivery of alerts has been enabled.



## **cma\_alert\_enable\_async**

The best application for asynchronous alert delivery is when the program needs to perform a long computation that does not affect program invariants and where adding calls to `cma_alert_test` is impractical or would slow down the computation.

Enabling asynchronous delivery over such a region of code allows the code to be highly responsive to abort requests without complicating or slowing the computation.

For example, a matrix multiply has only two real states: either the product was created successfully, or it was not. Generally, there is no need to know about the state of the product if it could not be completed. A matrix multiply can also take a very long time and should, therefore, be alertable. But because it involves several nested loops whose limits are parameters, it is not obvious where to place calls to `cma_alert_test` within it, or desirable to do so. To ensure that the matrix multiply is alertable, perform the following steps:

1. Set a flag indicating that the product is not available.
2. Disable asynchronous alerts to get the prior state.
3. Enable asynchronous delivery of alerts.
4. Perform multiplication (which may take a long time but is alertable).
5. Restore prior alert state.
6. Set a flag indicating the product is available.

See Example E-1 for more information.

## **Exceptions**

`cma_e_alerted`



---

## cma\_alert\_enable\_general

Enables general delivery of alerts to the current thread.

### Syntax

```
cma_alert_enable_general(  
    prior);
```

Argument	Data Type	Access
<i>prior</i>	opaque cma_t_alert_state	write

### C Binding

```
#include <cma.h>  
  
void  
cma_alert_enable_general (  
    cma_t_alert_state *prior);
```

### Arguments

#### **prior**

Receives the prior state of general alert delivery (enabled or disabled).

### Description

This routine enables general delivery of alerts to the current thread and returns the previous state of alert delivery to the *prior* argument. This routine is not alertable. A pending alert will remain pending until the next alert point.

### Exceptions

```
cma_e_alert_nesting  
cma_e_existence  
cma_e_use_error
```



## cma\_alert\_restore

---

### cma\_alert\_restore

Restores the state of general or asynchronous alert delivery.

#### Syntax

```
cma_alert_restore(  
    prior;
```

Argument	Data Type	Access
<i>prior</i>	opaque cma_t_alert_state	read

#### C Binding

```
#include <cma.h>  
  
void  
cma_alert_restore (  
    cma_t_alert_state *prior);
```

#### Arguments

##### **prior**

The prior state of alert delivery (enabled or disabled).

#### Description

This routine restores the alert delivery state (enabled or disabled) that was saved in the *prior* argument in a previous call to `cma_alert_disable_asynch`, `cma_alert_disable_general`, or `cma_alert_enable_general`.

For example, if general delivery of alerts is enabled when you call `cma_alert_disable_general`, the enabled setting is stored in *prior*. Subsequently calling `cma_alert_restore` and passing the *prior* value returned by `cma_alert_disable_general` causes the enabled setting to be restored, and general delivery of alerts is again enabled.

This routine is not alertable. Because this routine is not alertable, an alert that is already pending will remain pending after this call returns. If a pending alert should be delivered immediately, follow a call to this routine with a call to an alertable routine such as `cma_alert_test`.

Note that none of the general run-time routines are safe. None of the DECthreads library routines are safe except `cma_alert_restore` and `pthread_setasynccancel`.

#### Exceptions

```
cma_e_alert_nesting  
cma_e_badparam
```



---

## cma\_alert\_test

Requests delivery of a pending alert to the current thread.

### Syntax

```
cma_alert_test( );
```

### C Binding

```
#include <cma.h>

void
cma_alert_test (void);
```

### Arguments

None

### Description

This routine requests delivery of a pending alert to the current thread. The alert is delivered only if an alert is pending for the current thread and alert delivery is not currently disabled.

This routine is useful when called within very long loops to ensure that a pending alert is noticed within a reasonable amount of time.

This routine is alertable.

### Exceptions

cma\_e\_alerted



## cma\_attr\_create

---

## cma\_attr\_create

Creates an attributes object.

### Syntax

```
cma_attr_create(  
    new_attr,  
    attr);
```

Argument	Data Type	Access
new_attr	opaque cma_t_attr	write
attr	opaque cma_t_attr	read

### C Binding

```
#include <cma.h>  
  
void  
cma_attr_create (  
    cma_t_attr *new_attr,  
    cma_t_attr *attr);
```

### Arguments

#### new\_attr

Variable that receives the new attributes object.

#### attr

Handle of an attributes object used to specify attributes of the new attributes object. If you specify `cma_c_null` for the *attr* argument, default attributes are used.

### Description

This routine creates an attributes object that can be used to specify the attributes of DECthreads objects when they are created.

The individual attributes (internal fields) of the attributes object are set to default values. (The default values of each attribute are discussed in the descriptions of the following routines.) Use the following routines to change the individual attributes:

```
cma_attr_set_guardsize  
cma_attr_set_inherit_sched  
cma_attr_set_mutex_kind  
cma_attr_set_priority  
cma_attr_set_sched  
cma_attr_set_stacksize
```

When an attributes object is used to create an object (for example, a thread or mutex), the values of the individual attributes determine the characteristics of the new object. Attributes objects act like additional arguments to object creation. Changing individual attributes does not affect any objects that were previously created using the attributes object.



When you set the scheduling policy or priority, or both, in an attributes object, you must disable scheduling inheritance before the scheduling attributes are used.

## Exceptions

**cma\_e\_existence**  
**cma\_e\_use\_error**



## cma\_attr\_delete

---

### cma\_attr\_delete

Deletes an attributes object.

#### Syntax

```
cma_attr_delete(  
    attr);
```

Argument	Data Type	Access
attr	opaque cma_t_attr	modify

#### C Binding

```
#include <cma.h>  
  
void  
cma_attr_delete (  
    cma_t_attr    *attr);
```

#### Arguments

**attr**  
Handle of the attributes object deleted.

#### Description

This routine deletes an attributes object. Call this routine when an attributes object no longer needs to be referenced through the handle supplied by the `cma_attr_create` routine.

The effect of this routine is to give permission to reclaim storage for the attributes object. The attributes object is marked for deletion, and the *attr* argument is set to the value `cma_c_null`. Specifying `cma_c_null` for the *attr* argument is legal and has no effect. Objects that were created using this attributes object are not affected by the deletion of the attributes object.

The results of calling this routine are unpredictable if the handle specified by the *attr* argument refers to an attributes object that does not exist (unless it is `cma_c_null`.)

#### Exceptions

`cma_e_existence`  
`cma_e_use_error`



## cma\_attr\_get\_guardsize

Obtains the guardsize attribute of thread creation.

### Syntax

```
cma_attr_get_guardsize(
    attr,
    guardsize);
```

Argument	Data Type	Access
attr	opaque cma_t_attr	read
guardsize	opaque cma_t_natural	write

### C Binding

```
#include <cma.h>

void
cma_attr_get_guardsize (
    cma_t_attr *attr,
    cma_t_natural *guardsize);
```

### Arguments

#### attr

Handle of the attributes object whose guardsize attribute is obtained.

#### guardsize

Value of the guardsize attribute. The *guardsize* argument specifies the minimum size (in bytes) of the guard area for the stack of a thread.

### Description

This routine obtains the minimum size (in bytes) of the guard area for the stack of a thread that is created using the attributes object specified by the *attr* argument.

A guard area helps to detect stack overflows by preventing memory access beyond the thread's stack. Large guard areas are necessary when threads might allocate large structures on the stack.

### Exceptions

cma\_e\_existence  
cma\_e\_use\_error



## cma\_attr\_get\_inherit\_sched

---

## cma\_attr\_get\_inherit\_sched

Obtains the inherit scheduling attribute of thread creation.

### Syntax

```
cma_attr_get_inherit_sched(  
                                attr,  
                                setting);
```

Argument	Data Type	Access
attr	opaque cma_t_attr	read
setting	opaque cma_t_sched_inherit	write

### C Binding

```
#include <cma.h>  
  
void  
cma_attr_get_inherit_sched (  
    cma_t_attr *attr,  
    cma_t_sched_inherit *setting);
```

### Arguments

#### attr

Handle of the attributes object whose inherit scheduling attribute is obtained.

#### setting

Receives the value for the inherit scheduling attribute. Refer to the description of `cma_attr_set_inherit_sched` for valid values.

### Description

This routine obtains the inherit scheduling attribute of thread creation. The inherit scheduling attribute specifies whether threads created using the attributes object inherit the scheduling attributes of the creating thread, or use the scheduling attributes stored in the attributes object that is passed to `cma_thread_create`.

The default value of the inherit scheduling attribute is `cma_c_sched_inherit`.

### Exceptions

`cma_e_badparam`  
`cma_e_existence`  
`cma_e_use_error`



## cma\_attr\_get\_mutex\_kind

Obtains the mutex type attribute.

### Syntax

```
cma_attr_get_mutex_kind(
    attr,
    kind);
```

Argument	Data Type	Access
attr	opaque cma_t_attr	read
kind	opaque cma_t_mutex_kind	write

### C Binding

```
#include <cma.h>

void
cma_attr_get_mutex_kind (
    cma_t_attr *attr,
    cma_t_mutex_kind *kind);
```

### Arguments

#### attr

Handle of the attributes object whose mutex type is obtained.

#### kind

Value of the mutex type attribute. The *kind* argument specifies the type of mutex that is created. Valid values are `cma_c_mutex_fast` (default), `cma_c_mutex_recursive`, and `cma_c_mutex_nonrecursive`. (See Section 2.3.1 for definitions of the types of mutexes.)

### Description

This service obtains the mutex type attribute that is used when a mutex is created. See the `cma_attr_set_mutex_kind` description for information about mutex type attributes.

### Exceptions

```
cma_e_existence
cma_e_use_error
```



## cma\_attr\_get\_priority

Obtains the initial execution priority attribute of thread creation.

### Syntax

```
cma_attr_get_priority(
    attr,
    priority);
```

Argument	Data Type	Access
attr	opaque cma_t_attr	read
priority	opaque cma_t_priority	write

### C Binding

```
#include <cma.h>

void
cma_attr_get_priority (
    cma_t_attr *attr,
    cma_t_priority *priority);
```

### Arguments

#### attr

Handle of the attributes object whose priority attribute is obtained.

#### priority

Receives the value of the priority attribute. Refer to the description of `cma_attr_set_priority` for valid values.

### Description

This routine obtains the initial execution priority of threads created using the attributes object specified by the *attr* argument. The default value of the priority attribute is `cma_c_prio_through_mid`.

### Exceptions

```
cma_e_existence
cma_e_use_error
```



## cma\_attr\_get\_sched

Obtains the scheduling policy attribute of thread creation.

### Syntax

```
cma_attr_get_sched(
    attr,
    policy);
```

Argument	Data Type	Access
attr	opaque cma_t_attr	read
policy	opaque cma_t_sched_policy	write

### C Binding

```
#include <cma.h>

void
cma_attr_get_sched (
    cma_t_attr *attr,
    cma_t_sched_policy *policy);
```

### Arguments

#### attr

Handle of the attributes object whose scheduling policy attribute is obtained.

#### policy

Receives the value of the scheduling policy attribute. See the description of `cma_attr_set_sched` for valid values.

### Description

This routine obtains the scheduling policy of threads created using the attributes object specified by the *attr* argument. The default value of the scheduling attribute is `cma_c_sched_default` (which maps to `cma_c_sched_throughput`).

### Exceptions

`cma_e_existence`  
`cma_e_use_error`



## cma\_attr\_get\_stacksize

---

## cma\_attr\_get\_stacksize

Obtains the stacksize attribute of thread creation.

### Syntax

```
cma_attr_get_stacksize(  
    attr,  
    stacksize);
```

Argument	Data Type	Access
attr	opaque cma_t_attr	read
stacksize	opaque cma_t_natural	write

### C Binding

```
#include <cma.h>  
  
void  
cma_attr_get_stacksize (  
    cma_t_attr *attr,  
    cma_t_natural *stacksize);
```

### Arguments

#### attr

Handle of the attributes object whose stacksize attribute is obtained.

#### stacksize

Value of the stacksize attribute. The *stacksize* argument specifies the minimum size (in bytes) of the stack needed for a thread.

### Description

This routine obtains the minimum size (in bytes) of the stack needed for a thread created using the attributes object specified by the *attr* argument.

### Exceptions

cma\_e\_in\_use  
cma\_e\_use\_error



## cma\_attr\_set\_guardsize

Changes the guardsize attribute of thread creation.

### Syntax

```
cma_attr_set_guardsize(
    attr, guardsize);
```

Argument	Data Type	Access
attr	opaque cma_t_attr	read
guardsize	opaque cma_t_natural	read

### C Binding

```
#include <cma.h>

void
cma_attr_set_guardsize (
    cma_t_attr *attr,
    cma_t_natural guardsize);
```

### Arguments

**attr**  
Handle of the attributes object modified.

**guardsize**  
New value for the guardsize attribute. The *guardsize* argument specifies the minimum size (in bytes) of the guard area for the stack of a thread.

### Description

This routine sets the minimum size (in bytes) of the guard area for the stack of a thread that is created using the attributes object specified by the *attr* argument.

A guard area helps to detect stack overflows by preventing memory access beyond the thread's stack. Large guard areas might be necessary when threads allocate large structures on the stack.

### Exceptions

cma\_e\_badparam  
cma\_e\_existence  
cma\_e\_use\_error



## cma\_attr\_set\_inherit\_sched

---

### cma\_attr\_set\_inherit\_sched

Changes the inherit scheduling attribute of thread creation.

#### Syntax

```
cma_attr_set_inherit_sched(  
    attr,  
    setting);
```

Argument	Data Type	Access
attr	opaque cma_t_attr	read
setting	opaque cma_t_sched_inherit	read

#### C Binding

```
#include <cma.h>  
  
void  
cma_attr_set_inherit_sched (  
    cma_t_attr *attr,  
    cma_t_sched_inherit setting);
```

#### Arguments

**attr**  
Handle of the attributes object modified.

**setting**  
New value for the inherit priority attribute. Valid values are as follows:

cma_c_sched_inherit	This is the default value. The created thread inherits the current priority and scheduling policy of the thread calling <code>cma_thread_create</code> .
cma_c_sched_use_default	The created thread starts execution with the priority and scheduling policy stored in the attributes object.

#### Description

This routine changes the inherit scheduling attribute of thread creation. The inherit scheduling attribute specifies whether threads created using this attributes object inherit the scheduling attributes of the creating thread, or use the scheduling attributes stored in the attributes object that is passed to `cma_thread_create`.

The initial thread in an application, which is not created by a call to `cma_thread_create`, has an initial priority of `cma_c_prio_through_mid` and a scheduling policy of `cma_c_sched_other`. See the `cma_attr_set_priority` and `cma_attr_set_sched` routines for more information on valid priority values and valid scheduling policy values, respectively.



Inheriting scheduling attributes (instead of using the scheduling attributes stored in the attributes object) is useful when a thread is creating several helper threads—threads that are intended to work closely with the creating thread to cooperate in solving the same problem. For example, inherited scheduling attributes allow you to ensure that any helper threads created in a sort routine execute with the same priority as the calling thread.

---

#### Note

---

You must set scheduling inheritance to `cma_c_sched_use_default` if you want to create threads with a scheduling policy or priority different from the creating thread.

### Exceptions

`cma_e_badparam`  
`cma_e_existence`  
`cma_e_use_error`



---

### cma\_attr\_set\_mutex\_kind

Specifies the mutex type attribute.

#### Syntax

```
cma_attr_set_mutex_kind(  
    attr, kind);
```

Argument	Data Type	Access
attr	opaque cma_t_attr	read
kind	opaque cma_t_mutex_kind	read

#### C Binding

```
#include <cma.h>  
  
void  
cma_attr_set_mutex_kind (  
    cma_t_attr  *attr,  
    cma_t_mutex_kind  kind);
```

#### Arguments

##### attr

Handle of the attributes object modified.

##### kind

New value for the mutex type attribute. The *kind* argument specifies the type of mutex that is created. Valid values are `cma_c_mutex_fast` (default), `cma_c_mutex_recursive`, and `cma_c_mutex_nonrecursive`.

#### Description

This routine sets the mutex type attribute that is used when a mutex is created. A mutex can be fast, recursive, or nonrecursive. See Section 2.3.1 for more information on the types of mutexes.

#### Exceptions

`cma_e_badparam`  
`cma_e_existence`  
`cma_e_use_error`



## cma\_attr\_set\_priority

Changes the initial execution priority attribute of thread creation.

### Syntax

```
cma_attr_set_priority(
    attr,
    priority);
```

Argument	Data Type	Access
attr	opaque cma_t_attr	read
priority	opaque cma_t_priority	read

### C Binding

```
#include <cma.h>

void
cma_attr_set_priority (
    cma_t_attr  *attr,
    cma_t_priority  priority);
```

### Arguments

#### attr

Handle of the attributes object modified.

#### priority

New value for the priority attribute. The priority attribute is dependent upon scheduling policy. Valid values are as follows:

Low	Mid	High
cma_c_prio_fifo_min	cma_c_prio_fifo_mid	cma_c_prio_fifo_max
cma_c_prio_rr_min	cma_c_prio_rr_mid	cma_c_prio_rr_max
cma_c_prio_through_min	cma_c_prio_through_mid	cma_c_prio_through_max
cma_c_prio_back_min	cma_c_prio_back_mid	cma_c_prio_back_max

The default priority is cma\_c\_prio\_default\_mid. (This symbol maps to cma\_c\_prio\_through\_mid.)

### Description

This routine sets the initial execution priority of threads that were created using the attributes object specified by the *attr* argument. The default value of the priority attribute is cma\_c\_prio\_default\_mid.



## cma\_attr\_set\_priority

### Note

You must set scheduling inheritance to `cma_c_sched_use_default` if you want to create threads with a scheduling policy or priority different from the creating thread.

An application should specify priority only to express the urgency of executing the thread relative to other threads. Priority should not be used to control mutual exclusion when accessing shared data. With a sufficient number of processors executing, all ready threads, regardless of priority, can be executing simultaneously.

### Exceptions

`cma_e_badparam`  
`cma_e_existence`  
`cma_e_use_error`



## cma\_attr\_set\_sched

Changes the scheduling policy attribute of thread creation.

### Syntax

```
cma_attr_set_sched(
    attr,
    policy,
    priority);
```

Argument	Data Type	Access
attr	opaque cma_t_attr	read
policy	opaque cma_t_sched_policy	read
priority	opaque cma_t_priority	read

### C Binding

```
#include <cma.h>

void
cma_attr_set_sched (
    cma_t_attr *attr,
    cma_t_sched_policy policy,
    cma_t_priority priority);
```

### Arguments

#### attr

Handle of the attributes object modified.

#### policy

New value for the scheduling policy attribute. Valid values are as follows:

```
cma_c_sched_fifo
cma_c_sched_rr
cma_c_sched_throughput
cma_c_sched_background
```

See Section 2.2.3.2 for a description of the scheduling policies.

#### priority

New value for the priority attribute. The priority attribute is dependent upon scheduling policy. Valid values are as follows:

Low	Mid	High
cma_c_prio_fifo_min	cma_c_prio_fifo_mid	cma_c_prio_fifo_max
cma_c_prio_rr_min	cma_c_prio_rr_mid	cma_c_prio_rr_max
cma_c_prio_through_min	cma_c_prio_through_mid	cma_c_prio_through_max
cma_c_prio_back_min	cma_c_prio_back_mid	cma_c_prio_back_max



## cma\_attr\_set\_sched

The default priority is `cma_c_prio_default_mid`. (This symbol maps to `cma_c_prio_through_mid`.)

### Description

This routine sets the scheduling policy and priority of a thread that is created using the attributes object specified by the *attr* argument. The default value of the scheduling policy attribute is `cma_c_sched_default`. (This symbol is mapped to the `cma_c_sched_throughput` scheduling policy.) The default scheduling priority is the midrange of the default scheduling policy.

---

#### Note

---

You must set scheduling inheritance to `cma_c_sched_use_default` if you want to create threads with a scheduling policy or priority different from the creating thread.

---

### Exceptions

`cma_e_badparam`  
`cma_e_existence`  
`cma_e_use_error`



## cma\_attr\_set\_stacksize

Changes the stacksize attribute of thread creation.

### Syntax

```
cma_attr_set_stacksize(
    attr,
    stacksize);
```

Argument	Data Type	Access
attr	opaque cma_t_attr	read
stacksize	opaque cma_t_natural	read

### C Binding

```
#include <cma.h>

void
cma_attr_set_stacksize (
    cma_t_attr *attr,
    cma_t_natural stacksize);
```

### Arguments

**attr**  
Handle of the attributes object modified.

**stacksize**  
New value for the stacksize attribute. The *stacksize* argument specifies the minimum size (in bytes) of the stack needed for a thread.

### Description

This routine sets the minimum size (in bytes) of the stack needed for a thread created using the attributes object specified by the *attr* argument.

### Exceptions

```
cma_e_badparam
cma_e_existence
cma_e_stackovf
cma_e_use_error
```



## cma\_cond\_broadcast

---

## cma\_cond\_broadcast

Wakes all threads that are waiting on a condition variable.

### Syntax

```
cma_cond_broadcast(  
    condition);
```

Argument	Data Type	Access
condition	opaque cma_t_cond	read

### C Binding

```
#include <cma.h>  
  
void  
cma_cond_broadcast (  
    cma_t_cond    *condition);
```

### Arguments

#### condition

Handle of the condition variable broadcast.

### Description

This routine wakes all threads waiting on a condition variable. Calling this routine implies that data guarded by the associated mutex has changed so that it might be possible for more than one waiting thread to proceed. If only one waiting thread might be able to proceed, call `cma_cond_signal`.

You can call this routine when the associated mutex is either locked or unlocked.

### Exceptions

`cma_e_existence`  
`cma_e_use_error`



---

## cma\_cond\_create

Creates a condition variable.

### Syntax

```
cma_cond_create(
    new_condition,
    attr);
```

Argument	Data Type	Access
new_condition	opaque cma_t_cond	write
attr	opaque cma_t_attr	read

### C Binding

```
#include <cma.h>

void
cma_cond_create (
    cma_t_cond *new_condition,
    cma_t_attr *attr);
```

### Arguments

#### **new\_condition**

Variable that receives the handle for the new condition variable.

#### **attr**

Handle of the attributes object that defines the characteristics of the condition variable being created.

### Description

This routine creates and initializes a condition variable. A condition variable is a synchronization object used in conjunction with a mutex. A mutex controls access to shared data; a condition variable allows threads to wait for that data to enter a defined state. The state is defined by a predicate.

A condition variable can be signaled or broadcasted to indicate that a predicate might have become true. The broadcast routine indicates that all waiting threads should resume and reevaluate the predicate. The signal routine can be used in the special case where only one waiting thread can continue.

If a thread that holds a mutex determines that the shared data is not in the correct state for it to proceed (the associated predicate is not true), it can wait on a condition variable associated with the desired state. Waiting on the condition variable automatically releases the mutex so that other threads can modify or examine the shared data. When a thread modifies the state of the shared data so that a predicate might be true, it signals or broadcasts on the appropriate condition variable so that threads waiting for that predicate can continue.



## cma\_cond\_create

It is important that all threads waiting on a particular condition variable at any time hold the *same* mutex. At any time, an arbitrary number of condition variables can be associated with a single mutex, each representing a different predicate of the shared data protected by that mutex.

Condition variables are not owned by a particular thread. Any associated storage is not automatically deallocated when the creating thread terminates.

## Exceptions

**cma\_e\_existence**  
**cma\_e\_use\_error**



---

## cma\_cond\_delete

Deletes a condition variable.

### Syntax

```
cma_cond_delete(  
    condition);
```

Argument	Data Type	Access
condition	opaque cma_t_cond	modify

### C Binding

```
#include <cma.h>  
  
void  
cma_cond_delete (  
    cma_t_cond    *condition);
```

### Arguments

**condition**  
Handle of the condition variable deleted.

### Description

This routine deletes a condition variable. Call this routine when a condition variable is no longer referenced. The effect of calling this routine is to give permission to reclaim storage for the condition variable.

When the condition variable is deleted, the *condition* argument is set to the value `cma_c_null`. Specifying `cma_c_null` for *condition* is legal and has no effect.

The results of this routine are unpredictable and the `cma_e_existence` exception is raised if the handle specified in *condition* refers to a condition variable that does not currently exist (unless it is `cma_c_null`).

The results of this routine are unpredictable and the `cma_e_in_use` exception is raised if there are threads waiting for the specified condition variable to be signaled or broadcasted when it is deleted.

### Exceptions

`cma_e_existence`  
`cma_e_in_use`  
`cma_e_use_error`



## cma\_cond\_signal

---

## cma\_cond\_signal

Wakes one thread that is waiting on a condition variable.

### Syntax

```
cma_cond_signal(  
    condition);
```

Argument	Data Type	Access
condition	opaque cma_t_cond	read

### C Binding

```
#include <cma.h>  
  
void  
cma_cond_signal (  
    cma_t_cond *condition);
```

### Arguments

**condition**  
Handle of the condition variable signaled.

### Description

This routine wakes one thread waiting on a condition variable. Calling this routine implies that data guarded by the associated mutex has changed so that it might be possible for a single waiting thread to proceed. Call this routine when any thread waiting on the specified condition variable might find its predicate true, but only one thread should proceed.

You can call this routine when the associated mutex is either locked or unlocked.

If you want to signal a thread from interrupt level, use `cma_cond_signal_int`.

Do not call this routine from any software interrupt handler.

### Exceptions

`cma_e_existence`  
`cma_e_use_error`



---

## cma\_cond\_signal\_int

**Wakes one thread that is waiting on a condition variable. This routine can only be called from interrupt level.**

### Syntax

```
cma_cond_signal(  
    condition);
```

Argument	Data Type	Access
condition	opaque cma_t_cond	read

### C Binding

```
#include <cma.h>  
  
void  
cma_cond_signal_int (  
    cma_t_cond    *condition);
```

### Arguments

**condition**  
Handle of the condition variable signaled.

### Description

This routine wakes one thread waiting on a condition variable. It can only be called from a software interrupt handler routine. Calling this routine implies that it might be possible for a single waiting thread to proceed. Call this routine when any thread waiting on the specified condition variable might find its predicate true.

This routine does not cause a thread blocked on a condition variable to resume execution immediately. A thread resumes execution after the interrupt handler returns.

You can call this routine when the associated mutex is either locked or unlocked. (Note that you should never try to lock a mutex from an interrupt handler.)

---

#### Note

This routine allows you to signal a thread from a software interrupt handler. Do not call this routine from noninterrupt code. If you want to signal a thread from the normal noninterrupt level, use `cma_cond_signal`.

---

### Exceptions

`cma_e_existence`  
`cma_e_use_error`



---

### cma\_cond\_timed\_wait

Causes a thread to wait for a condition variable to be signaled or broadcasted for a specified period of time.

#### Syntax

```
cma_cond_timed_wait(  
    condition,  
    mutex,  
    expiration);
```

Argument	Data Type	Access
condition	opaque cma_t_cond	read
mutex	opaque cma_t_mutex	read
expiration	opaque cma_t_date_time	read

#### C Binding

```
#include <cma.h>  
  
cma_t_status  
cma_cond_timed_wait (  
    cma_t_cond *condition,  
    cma_t_mutex *mutex,  
    cma_t_date_time *expiration);
```

#### Arguments

##### condition

Handle of the condition variable on which the thread waits.

##### mutex

Mutex associated with the condition variable specified in *condition*.

##### expiration

Absolute time at which the wait should expire, if the condition has not yet been signaled or broadcasted. (See the *cma\_time\_get\_expiration* routine, which can be used to obtain a value for this argument.)

#### Description

This routine causes a thread to wait until:

- A condition variable is signaled or broadcasted.
- The current system clock time is greater than or equal to the time specified by the *expiration* argument.

This routine is identical to *cma\_cond\_wait* except that this routine can return before a condition variable is signaled or broadcasted, specifically, when a specified time expires.



Valid return values are `cma_s_normal` and `cma_s_timed_out`. If the wait is completed normally by a signal or broadcast on the condition variable, the return status is `cma_s_normal`. If the wait completes because the expiration time has passed, the return status is `cma_s_timed_out`.

If the current time equals or exceeds the expiration time, this routine returns immediately, without causing the current thread to wait. Your code should check the return status whenever this routine returns and take the appropriate action. Otherwise, waiting on the condition variable can become a nonblocking loop.

Call this routine after you have locked the mutex specified in *mutex*. The results of this routine are unpredictable if this routine is called without first locking the mutex.

## Exceptions

`cma_e_alerted`  
`cma_e_existence`  
`cma_e_use_error`



## cma\_cond\_wait

---

### cma\_cond\_wait

Causes a thread to wait for a condition variable to be signaled or broadcasted.

#### Syntax

```
cma_cond_wait(  
    condition,  
    mutex);
```

Argument	Data Type	Access
condition	opaque cma_t_cond	read
mutex	opaque cma_t_mutex	read

#### C Binding

```
#include <cma.h>  
  
void  
cma_cond_wait (  
    cma_t_cond *condition,  
    cma_t_mutex *mutex);
```

#### Arguments

##### condition

Handle of the condition variable on which the thread waits.

##### mutex

Mutex associated with the condition variable specified in *condition*.

#### Description

This routine causes a thread to wait for a condition variable to be signaled or broadcasted. Each condition corresponds to one or more predicates upon the data. The calling thread waits for the data to reach a particular state (for a particular predicate to become true).

Call this routine after you have locked the mutex specified in *mutex*. The results of this routine are unpredictable if this routine is called without first locking the mutex.

This routine atomically releases the mutex and causes the calling thread to wait on the condition. If the wait is satisfied as a result of some thread calling `cma_cond_signal` or `cma_cond_broadcast`, the mutex is reacquired and the routine returns.

As a general rule, a thread that has changed the state of storage protected by the mutex in such a way that a predicate associated with a condition variable might now be true must call either `cma_cond_signal` or `cma_cond_broadcast` for that condition variable. If neither call is made, any thread waiting on the condition variable continues to wait.

This routine is alertable. Alertable means that a pending alert is noticed during the wait on the condition variable. This helps ensure that all long waits can be canceled by alerting the thread.



If the `cma_e_alerted` exception is raised, the mutex is reacquired before the exception is raised.

This routine might (with low probability) return when the condition variable has not been signaled or broadcasted. When a spurious wakeup occurs, the mutex is reacquired before the routine returns. (To handle this type of situation, this routine should always be enclosed in a loop that tests for the desired shared data state.)

## Exceptions

`cma_e_alerted`  
`cma_e_existence`  
`cma_e_use_error`



## **cma\_debug**

---

### **cma\_debug**

Invokes the DECthreads internal debugger.

#### **Syntax**

```
cma_debug( );
```

#### **C Binding**

```
#include <cma.h>
```

```
void
```

```
cma_debug (void);
```

#### **Arguments**

None

#### **Description**

This routine invokes the DECthreads internal debugger as a callable function. It takes no arguments and does not return a value. It enters the internal debugger parsing loop. Type exit to return to the program.

To pass a list of debugging commands to DECthreads, call `cma_debug_cmd`. For more information, see Appendix D.

#### **Exceptions**

None



## cma\_debug\_cmd

Passes a list of cma\_debug commands to DECthreads.

### Syntax

```
cma_debug_cmd(
    cmd);
```

Argument	Data Type	Access
cmd	character string	read

### C Binding

```
#include <cma.h>

cma_t_dbg_status
cma_debug_cmd (
    char *cmd);
```

### Arguments

**cmd**  
cma\_debug command string. Null terminated string, commands separated by semicolons.

### Description

This routine passes a list of debugging commands to DECthreads. Each command is executed in sequence. Any output is written to standard output. This routine returns when the final command (or Exit command) is executed.

For a list of cma\_debug commands, see Appendix D.

Following are two examples of calling this routine:

```
cma_debug_cmd ("thread -b; mu -lq; cond -wq");
cma_debug_cmd ("att");
```

If you want to invoke the debugger for interactive commands, call cma\_debug.

### Exceptions

None



---

## cma\_delay

Causes a thread to wait for a specified period of time before continuing execution.

### Syntax

```
cma_delay(  
    interval);
```

Argument	Data Type	Access
time_interval	single precision floating point	read

### C Binding

```
#include <cma.h>  
  
void  
cma_delay (  
    cma_t_interval    interval);
```

### Arguments

#### interval

Number of seconds that the calling thread waits before continuing execution. Specify a value greater than or equal to 0.

### Description

This routine causes a thread to delay execution for a specified period of elapsed time. The period of time the thread waits is at least as long as the number of seconds specified in the *time\_interval* argument.

If you specify a value for *time\_interval* that is less than 0, the `cma_e_badparam` exception is raised. Specifying 0 for *time\_interval* is allowed and can result in the thread giving up the processor or delivering a pending alert.

This routine is alertable.

### Exceptions

```
cma_e_alerted  
cma_e_badparam
```



## cma\_handle\_assign

Assigns a handle to an object.

### Syntax

```
cma_handle_assign(  
    handle1,  
    handle2);
```

Argument	Data Type	Access
handle1	opaque cma_t_handle	read
handle2	opaque cma_t_handle	write

### C Binding

```
#include <cma.h>  
  
void  
cma_handle_assign (  
    cma_t_handle *handle1,  
    cma_t_handle *handle2);
```

### Arguments

#### handle1

Handle that is assigned to *handle2*.

#### handle2

Handle that receives the value from *handle1*.

### Description

This routine assigns the value of a handle, or name, to an object. Handles are allocated by the user application. This routine allows you to copy a handle from one object to another.

When an object is created, the storage for the object is allocated and initialized, and a handle for the object is returned. The handle is the only means of referring to an object and performing routines on the object. Because objects are only accessed through handles, you can usually think of the handle as if it were the object itself.

Handles are meaningful only within a single process address space. Attempting to access an object from a process other than the one in which it was created—for example, by means of multiply mapped memory—can result in unpredictable results (it is incorrect, but the error is not necessarily checked).

### Exceptions

None



## cma\_handle\_equal

---

## cma\_handle\_equal

Compares one handle to another handle.

### Syntax

```
cma_handle_equal(  
    handle1,  
    handle2);
```

Argument	Data Type	Access
handle1	opaque cma_t_handle	read
handle2	opaque cma_t_handle	read

### C Binding

```
#include <cma.h>  
  
cma_t_boolean  
cma_handle_equal (  
    cma_t_handle *handle1,  
    cma_t_handle *handle2);
```

### Arguments

#### handle1

The first handle to be compared.

#### handle2

The second handle to be compared.

### Description

This routine compares one handle to another handle. (This routine does not check whether the objects that correspond to the handles currently exist.) The value `cma_c_true` is returned if the handles have values indicating that they designate the same object. If the values do not designate the same object, the value `cma_c_false` is returned.

### Exceptions

None



---

**cma\_init**

Initializes the DECthreads routines.

**Syntax**

```
cma_init( );
```

**C Binding**

```
#include <cma.h>
```

```
void  
cma_init (void);
```

**Arguments**

None

**Description**

This routine initializes the internal storage and process-wide state that is necessary to support DECthreads routines.

---

**Note**

---

On ULTRIX operating systems, call this routine before calling any other cma routine. On other operating systems this routine has no effect.

---

Calling this routine can result in changes to the execution environment that cause a single-threaded process to appear as a DECthreads thread.

If this routine is called more than once, the second and subsequent calls are ignored. If a second call to this routine is made while the first call is still in progress, the exception `cma_e_inialrpro` is raised.

**Exceptions**

`cma_e_inialrpro`



## cma\_key\_create

---

## cma\_key\_create

Generates a unique per-thread context key value.

### Syntax

```
cma_key_create(  
    key,  
    attr,  
    destructor);
```

Argument	Data Type	Access
key	opaque cma_t_key	write
attr	opaque cma_t_attr	read
destructor	procedure cma_t_destructor	read

### C Binding

```
#include <cma.h>  
  
void  
cma_key_create (  
    cma_t_key    *key,  
    cma_t_attr   *attr,  
    cma_t_destructor destructor);
```

### Arguments

#### key

Receives the value of the new per-thread context key.

#### attr

Handle of the attributes object that defines the characteristics of the per-thread context key being created.

#### destructor

Procedure called to destroy a context value associated with this key when the thread terminates.

### Description

This routine generates a unique per-thread context key value. This key value identifies a per-thread context, which is an address of memory generated by the client containing arbitrary data of any size.

Per-thread context is a mechanism that allows client software to associate context information with the current thread. (This mechanism can be thought of as a means for a client to add its own unique fields to the thread control block.)

This routine generates and returns a new key value. Each call to this routine within a process returns a key value that is unique within an application invocation. Keys must be generated from initialization code that is guaranteed to be called only once within each process. (Refer to the description of `cma_once` for more information.)



When multiple facilities share access to per-thread context, the facilities must agree on the key value that is associated with the context. The key value must be created only once, and should be stored in a location known to each facility. (It may be desirable to encapsulate the creation of a key, and the setting and getting of context values for that key, within a special facility created for that purpose.)

When a thread terminates, per-thread context is automatically destroyed. For each per-thread context currently associated with the thread, the destructor routine associated with the key value of that context is called.

## Exceptions

**cma\_e\_existence**

**cma\_e\_use\_error**



## cma\_key\_get\_context

---

### cma\_key\_get\_context

Obtains the per-thread context associated with the specified key.

#### Syntax

```
cma_key_get_context(  
    key,  
    context_value);
```

Argument	Data Type	Access
key	opaque cma_t_key	read
context_value	opaque cma_t_address	write

#### C Binding

```
#include <cma.h>  
  
void  
cma_key_get_context (  
    cma_t_key    key,  
    cma_t_address *context_value);
```

#### Arguments

##### key

Context key value that uniquely identifies the context value obtained. This key value must have been obtained from `cma_key_create`.

##### context\_value

Variable that receives the address of the current per-thread context value associated with the specified key.

#### Description

This routine obtains the per-thread context associated with the specified key for the current thread. If a context has not been defined for the key in this thread, the null pointer `cma_c_null_ptr` is returned in *context\_value*.

The exception `cma_e_badparam` is raised if the context key is invalid.

#### Exceptions

`cma_e_badparam`



---

## cma\_key\_set\_context

Sets the per-thread context associated with the specified key for the current thread.

### Syntax

```
cma_key_set_context(
    key,
    context_value);
```

Argument	Data Type	Access
key	opaque cma_t_key	read
context_value	opaque cma_t_address	read

### C Binding

```
#include <cma.h>

void
cma_key_set_context (
    cma_t_key    key,
    cma_t_address context_value);
```

### Arguments

#### key

Context key value that uniquely identifies the context value specified in *context\_value*. This key value must have been obtained from *cma\_key\_create*.

#### context\_value

Address containing data associated with the specified key for the current thread; this is the per-thread context.

### Description

This routine sets the per-thread context associated with the specified key for the current thread. If a context has been defined for the key in this thread (the current value is not null), the new value is substituted for it.

The exception *cma\_e\_badparam* is raised if the context key is invalid.

### Exceptions

*cma\_e\_badparam*



## **cma\_lock\_global**

---

### **cma\_lock\_global**

Locks the global mutex.

#### **Syntax**

```
cma_lock_global( );
```

#### **C Binding**

```
#include <cma.h>

void
cma_lock_global (void);
```

#### **Arguments**

None

#### **Description**

This routine locks the global mutex. If the global mutex is currently locked by another thread when a thread calls this routine, the calling thread waits for the global mutex to become available.

The thread that has locked the global mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the global mutex in the locked state and with the current thread as the global mutex's current owner.

Use the global mutex when calling a library package that is not designed to run in a multithreaded environment. (Unless the documentation for a library function specifically states that it is compatible with multithreading, assume that it is not compatible; in other words, assume it is nonreentrant.)

The global mutex is one lock. Any code that calls any function that isn't known to be reentrant uses the same lock. This prevents dependencies among threads calling library functions and those functions calling other functions, and so on.

The global mutex is a recursive mutex. A thread that has locked the global mutex can relock it without deadlocking. (The locking thread must call `cma_unlock_global` as many times as it called this routine to allow another thread to lock the global mutex.)

Do not call this routine from any software interrupt handler.

#### **Exceptions**

```
cma_e_existence
cma_e_use_error
```



---

## cma\_mutex\_create

Creates a mutex.

### Syntax

```
cma_mutex_create(
    new_mutex,
    attr);
```

Argument	Data Type	Access
new_mutex	opaque cma_t_mutex	write
attr	opaque cma_t_attr	read

### C Binding

```
#include <cma.h>

void
cma_mutex_create (
    cma_t_mutex *new_mutex,
    cma_t_attr *attr);
```

### Arguments

#### **new\_mutex**

Receives a handle for the mutex.

#### **attr**

Handle of the attributes object that defines the characteristics of the mutex created. If you specify `cma_c_null`, default attributes are used.

### Description

This routine creates a mutex. A mutex is a synchronization object that allows multiple threads to serialize their access to shared data.

The mutex is created and initialized to the unlocked state.

If the thread that called this routine terminates, the created mutex is not automatically deallocated because it is considered to be shared among multiple threads.

### Exceptions

`cma_e_existence`  
`cma_e_use_error`



## cma\_mutex\_delete

---

## cma\_mutex\_delete

Deletes a mutex.

### Syntax

```
cma_mutex_delete(  
    mutex);
```

Argument	Data Type	Access
mutex	opaque cma_t_mutex	modify

### C Binding

```
#include <cma.h>  
  
void  
cma_mutex_delete (  
    cma_t_mutex *mutex);
```

### Arguments

#### mutex

Handle of the mutex deleted. After the call to this routine, the *mutex* argument is set to the value `cma_c_null`.

### Description

This routine deletes a mutex and should be called when a mutex is no longer referenced. Calling this routine reclaims storage for the mutex object.

After the mutex is deleted, the *mutex* argument is set to the value `cma_c_null`. Calling this routine and specifying a value of `cma_c_null` for the *mutex* argument is legal and has no effect.

Do not delete a mutex that has a current owner (in other words, is locked). If you try to delete a mutex that is locked, the `cma_e_in_use` exception is raised.

The results of this routine are unpredictable if the handle specified in the *mutex* argument refers to a mutex object that does not currently exist (unless it is `cma_c_null`).

### Exceptions

`cma_e_existence`  
`cma_e_in_use`  
`cma_e_use_error`



## cma\_mutex\_lock

Locks a mutex if the mutex is unlocked. If the mutex is locked, causes the thread to wait for the mutex to become available.

### Syntax

```
cma_mutex_lock(  
    mutex);
```

Argument	Data Type	Access
mutex	opaque cma_t_mutex	read

### C Binding

```
#include <cma.h>  
  
void  
cma_mutex_lock (  
    cma_t_mutex *mutex);
```

### Arguments

**mutex**  
Handle of the mutex locked.

### Description

This routine locks a mutex and its behavior varies based on the kind of mutex.

If you specified a recursive mutex, the current owner of a mutex can relock the same mutex without blocking. If you specified a nonrecursive mutex and the current owner tries to lock the mutex a second time, the exception `cma_e_in_use` is reported. If the mutex is locked when a thread calls this routine, the thread waits for the mutex to become available. If you specified a fast mutex, a deadlock can result if the current owner of a mutex calls this routine in an attempt to lock the mutex a second time.

The thread that has locked a mutex becomes its current owner and remains the owner until the same thread (and only that thread) has unlocked it. This routine returns with the mutex in the locked state and with the current thread as the mutex's current owner.

See `cma_attr_set_mutex_kind` for information about fast, recursive, and nonrecursive mutexes.

### Exceptions

`cma_e_existence`  
`cma_e_in_use`  
`cma_e_use_error`



## cma\_mutex\_try\_lock

---

### cma\_mutex\_try\_lock

Locks a mutex. If the mutex is already locked, the calling thread does not wait for the mutex to become available.

#### Syntax

```
cma_mutex_try_lock(  
    mutex);
```

Argument	Data Type	Access
mutex	opaque cma_t_mutex	read

#### C Binding

```
#include <cma.h>  
  
cma_t_boolean  
cma_mutex_try_lock (  
    cma_t_mutex *mutex);
```

#### Arguments

##### mutex

Handle of the mutex to be locked.

#### Description

This routine locks a mutex and its behavior varies based on the kind of mutex. If the specified mutex is locked when a thread calls this routine, the calling thread does not wait for the mutex to become available.

When a thread calls this routine, an attempt is made to immediately lock the mutex. If the mutex is successfully locked, the Boolean value `cma_c_true` is returned. The current thread is then the mutex's current owner.

If the mutex is already locked when this routine is called (even if it was previously locked by the current thread), the Boolean value `cma_c_false` is returned and the thread does not wait to acquire the lock.

#### Exceptions

`cma_e_existence`  
`cma_e_use_error`



## cma\_mutex\_unlock

Unlocks a mutex.

### Syntax

```
cma_mutex_unlock(  
    mutex);
```

Argument	Data Type	Access
mutex	opaque cma_t_mutex	read

### C Binding

```
#include <cma.h>  
  
void  
cma_mutex_unlock (  
    cma_t_mutex *mutex);
```

### Arguments

**mutex**  
Handle of the mutex unlocked.

### Description

This routine unlocks a mutex and its behavior varies based on the kind of mutex. When an owner unlocks a recursive mutex, the lock count is decremented. The mutex remains locked and owned until the count reaches zero. When the lock count reaches 0, or for any other type of mutex, the mutex becomes unlocked with no current owner. If one or more threads are waiting to lock the specified mutex, calling this routine causes one thread to unblock and try to acquire the mutex.

The results of calling this routine are unpredictable if the mutex specified in *mutex* is already unlocked. In that case, the exception `cma_e_use_error` is raised.

The results of calling this routine are also unpredictable if the mutex specified in *mutex* is currently owned by a thread other than the calling thread. In that case, the exception `cma_e_in_use` is raised.

### Exceptions

`cma_e_existence`  
`cma_e_in_use`  
`cma_e_use_error`



---

## cma\_once

Calls an initialization routine that can be executed by only one thread, a single time.

### Syntax

```
cma_once(
    init_block,
    init_routine,
    arg);
```

Argument	Data Type	Access
init_block	opaque cma_t_once	read
init_routine	opaque cma_t_init_routine	read
arg	opaque cma_t_address	read

### C Binding

```
#include <cma.h>

void
cma_once (
    cma_t_once *init_block,
    cma_t_init_routine *init_routine,
    cma_t_address arg);
```

### Arguments

#### init\_block

Address of a record that defines the one-time initialization code. Each one-time initialization routine must have its own unique *cma\_t\_once*.

#### init\_routine

Address of a procedure that performs the initialization. This routine is called only once, regardless of the number of times it and its associated *init\_block* are passed to *cma\_once*.

#### arg

Argument passed to the *init\_routine*.

### Description

This routine calls an initialization routine that can be executed by only one thread, a single time. This routine allows you to create your own initialization code that is guaranteed to be run only once, even if called simultaneously by multiple threads.

For example, a mutex or a per-thread context key must be created exactly once. Calling *cma\_once* prevents the problem that occurs when the code that creates a mutex or per-thread context can be called by multiple threads. Without this routine, the execution must be serialized so that only one thread performs the initialization. Other threads that reach the same point in the code would be delayed until the first thread is finished.



This routine initializes the control record if it has not already been initialized, and then determines if the client one-time initialization routine has already executed once. If it has not executed, then this routine calls the initialization routine specified in *init\_routine*. If the client one-time initialization code has already executed once, then this routine returns.

Because the *init\_routine* accepts an argument (*arg*), a single initialization routine can be used to initialize any number of objects. For example, an initialization routine that creates a global mutex might take the address of a *cma\_t\_mutex* variable, which receives the handle of a new mutex. Note that you cannot make every call to the initialization routine using the same control block; it would only be called once. Effectively, each value of *arg* must be associated with its own control block.

---

#### Note

---

If you specify an *init\_routine* that directly or indirectly results in a recursive call to *cma\_once* specifying the same *init\_block* argument, the recursive call will result in a deadlock.

---

The *init\_block* must be declared static (for example, either extern or static in the C language), and it must be initialized at compile time. In the C language, using *cma.h*, initialize an *init\_block* using the *cma\_once\_init* macro. In other languages, you must initialize a *cma\_t\_once* block to a value of three integer zeros. In C, that corresponds to the following:

```
static cma_t_once block = {0,0,0};
```

The result of this routine is unpredictable and the *cma\_e\_badparam* exception is raised if the *init\_block* is not a properly initialized one-time initialization block.

## Exceptions

*cma\_e\_badparam*

## Example

The following C code segment declares a one-time initialization section that creates a mutex for later use:

```
#include <CMA.H>
static cma_t_once make_my_mutex = cma_once_init; ❶
static cma_t_mutex my_mutex; ❷
.
.
.
void initialize_mutex (cma_t_address_arg) { ❸
    cma_mutex_create (&my_mutex, &cma_c_null);
}
.
.
.
cma_once (&make_my_mutex, initialize_mutex, 0); ❹
```

- ❶ Declare the *cma\_t\_once* that defines the particular one-time initialization code.



## cma\_once

- 2 Declare the mutex to be initialized.
- 3 Declare the initialization routine that will create the mutex.
- 4 Call `cma_once` with the initialization control block and routine. If no thread has already executed the initialization routine, it will be called. Otherwise, `cma_once` returns.



---

## cma\_stack\_check\_limit\_np

Determines whether sufficient space exists on the current thread's stack to allocate the requested number of bytes of local storage.

### Syntax

```
cma_stack_check_limit_np(
    size);
```

Argument	Data Type	Access
size	opaque cma_t_integer	read

### C Binding

```
#include <cma.h>

cma_t_boolean
cma_stack_check_limit_np (
    cma_t_integer  size);
```

### Arguments

#### size

Number of bytes requested. DECThreads determines whether a stack allocation of the specified size extends beyond the end of the thread's stack.

### Description

This routine determines whether sufficient space exists on the current thread's stack to allocate the requested number of bytes of local storage. If the requested size fits (if it does not extend beyond the current thread's stack), `cma_c_true` is returned. If the requested size extends beyond the end of the stack, `cma_c_false` is returned.

A DECThreads stack consists of the following three parts:

- A **green zone**, which is the normal area where procedure activation frames and stack automatic variables are allocated.
- A **reserved zone**, which is available for allocation but indicates that the thread has almost reached the end of the stack. (A thread should not use the reserved zone; it has been set aside for use by error-handling mechanisms.)
- A **guard zone**, which is normally protected with no access so that an attempt by the thread to read or write will fail with a hardware error.

If the thread does not use the `cma_stack_check_limit_np` routine, it is possible for a thread to skip over the guard zone of its stack by, for example, allocating a very large array on the stack. If the thread writes to the part of the array that extends beyond its own guard zone before attempting to access the part of the array in its own guard zone, it would corrupt the memory allocated at that location; for example, another thread's stack. This results in unpredictable behavior of the application and is difficult to debug. The `cma_stack_check_limit_np` routine recognizes that situation and returns `cma_c_false`.



## cma\_stack\_check\_limit\_np

When `cma_stack_check_limit_np` is called from the thread that is running on the default process stack, `cma_stack_check_limit_np` attempts to access each page that would fall within the requested allocation. It returns `cma_c_true` unless the system is unable to expand the process stack to the needed size.

### Returns

Returns a Boolean value that specifies whether the requested size can be allocated without overflowing the current thread's stack. Possible values are:

`cma_c_true`

Space can be allocated.

`cma_c_false`

Space cannot be allocated.

### Exceptions

None



## cma\_thread\_alert

Allows a thread to request that it or another thread terminate execution.

### Syntax

```
cma_thread_alert(
    thread);
```

Argument	Data Type	Access
thread	opaque cma_t_thread	read

### C Binding

```
#include <cma.h>

void
cma_thread_alert (
    cma_t_thread *thread);
```

### Arguments

**thread**  
Handle of the thread that receives an alert.

### Description

This routine sends an alert to the specified thread. Issuing an alert does not guarantee that the alerted thread will receive or handle the alert. The alerted thread can delay processing the alert after receiving it. For example, if the alerted thread has disabled alerts during an important operation, it will continue because it cannot be interrupted at the point where the alert is requested.

Because of communication delays, the calling thread can only rely on the fact that an alert will eventually become pending in the designated thread (provided that the thread does not terminate beforehand). Furthermore, the calling thread has no guarantee that a pending alert will be delivered because delivery is controlled by the designated thread.

The `cma_e_existence` exception is raised if the value specified in *thread* is `cma_c_null`, or if it refers to a thread that does not currently exist.

This routine is not alertable.

### Exceptions

`cma_e_existence`  
`cma_e_use_error`



## cma\_thread\_bind\_to\_cpu

### cma\_thread\_bind\_to\_cpu

Binds a thread to a particular CPU on a multiprocessor system.

#### Syntax

```
cma_thread_bind_to_cpu(  
    thread,  
    cpu_mask);
```

Argument	Data Type	Access
thread	opaque cma_t_thread	read
cpu_mask	unsigned long	read

#### C Binding

```
#include <cma.h>  
  
void  
cma_thread_bind_to_cpu (  
    cma_t_thread *thread,  
    unsigned long cpu_mask);
```

#### Arguments

##### thread

Handle of the thread to bind to the specified processor.

##### cpu\_mask

Bit mask specifying which CPU the thread is to be bound to. The low-order bit represents the first CPU. Only one bit in this mask may be set.

#### Description

This routine binds a thread to a particular processor on a multiprocessor system. Once bound, a thread will only run on the specified CPU until the thread terminates, or the binding is changed.

Specify a *cpu\_mask* of 0 to allow a previously bound thread to execute on any available CPU.

This routine is not available on all platforms. If it is not available, it will raise the *cma\_e\_unimp* exception.

#### Exceptions

*cma\_e\_badparam*  
*cma\_e\_unimp*



## cma\_thread\_create

Creates a thread object and thread.

### Syntax

```
cma_thread_create(
    new_thread,
    attr,
    start_routine,
    arg);
```

Argument	Data Type	Access
new_thread	opaque cma_t_thread	write
attr	opaque cma_t_attr	read
start_routine	cma_t_start_routine	read
arg	pointer	read

### C Binding

```
#include <cma.h>
void
cma_thread_create (
    cma_t_thread  *new_thread,
    cma_t_attr    *attr,
    cma_t_start_routine  start_routine,
    cma_t_address  arg);
```

### Arguments

#### new\_thread

Variable that receives a handle for the thread object.

#### attr

Handle of the attributes object that defines the characteristics of the thread being created. If you specify `cma_c_null`, default attributes are used.

#### start\_routine

Function executed as the new thread's start routine. This argument is the address of a routine that takes one argument of type `cma_t_address`, and returns a value of type `cma_t_address`.

#### arg

Address value that is copied and passed to the thread's start routine.



## cma\_thread\_create

### Description

This routine creates a thread object and a thread. The thread routine is a function of type `cma_t_start_routine`. The function accepts a single argument of type `cma_t_address` and returns a function value of type `cma_t_address`. For example, the following routine coded in Ada, is compatible with the `cma_t_start_routine` type:

```
function START_ROUTINE (  
    ARG : in CMA_T_ADDRESS) return CMA_T_ADDRESS;
```

The same example coded in C, is as follows:

```
cma_t_address  
start_routine (  
    cma_t_address arg);
```

Calling this routine sets into motion the following actions:

- An internal thread object is created to describe the thread.
- The associated executable thread is created with attributes specified by the *attr* argument (or with default attributes if `cma_c_null` is specified.)
- The *new\_thread* argument receives the handle of the new thread.
- The *start\_routine* function is called.

The thread is created in the ready state and therefore might immediately begin executing the function specified by the *start\_routine* argument. The newly created thread will begin running before `cma_thread_create` completes if the new thread follows the `cma_c_sched_rr` or `cma_c_sched_fifo` scheduling policy or has a priority higher than the creating thread, or both. Otherwise, the new thread begins running at its turn, which might also be before `cma_thread_create` returns.

The *start\_routine* is passed a copy of the *arg* argument. The value of the *arg* argument is specified by the calling application code.

The thread object exists until the `cma_thread_detach` routine is called and the thread terminates, whichever occurs last.

Synchronization between the caller of `cma_thread_create` and the newly created thread is done through the use of the `cma_thread_join` routine (or any other mutexes or condition variables they agree to use).

### Exceptions

`cma_e_existence`  
`cma_e_use_error`



## cma\_thread\_detach

Marks a thread object for deletion.

### Syntax

```
cma_thread_detach(  
    thread);
```

Argument	Data Type	Access
thread	opaque cma_t_thread	modify

### C Binding

```
#include <cma.h>  
  
void  
cma_thread_detach (  
    cma_t_thread *thread);
```

### Arguments

#### thread

Handle of the thread object marked for deletion.

### Description

This routine indicates that storage for the specified thread can be reclaimed when the thread terminates. If the thread object is no longer needed by the thread, then the thread object is deallocated immediately. The *thread* argument is set to the value `cma_c_null`.

Call this routine when no other threads are interested in joining with the thread. Call this routine where appropriate for every thread that is created to ensure that storage for thread objects does not accumulate.

Once this routine has been called, other threads cannot join with the detached thread.

Calling this routine for a value for *thread* of `cma_c_null` is legal and has no effect.

The results of this routine are unpredictable if the value of *thread* refers to a thread object that does not currently exist.

### Exceptions

`cma_e_existence`  
`cma_e_use_error`



## cma\_thread\_exit\_error

---

## cma\_thread\_exit\_error

Terminates the current thread when an error occurs.

### Syntax

```
cma_thread_exit_error( );
```

### C Binding

```
#include <cma.h>

void
cma_thread_exit_error(void);
```

### Arguments

None

### Description

This routine terminates execution of the current thread within an arbitrary routine when an error occurs. Normally, a thread terminates when the *start\_routine* argument to *cma\_thread\_create* returns.

Call this routine only when an error occurs that requires thread termination and you do not want to signify the error by raising an exception. (Raising an exception is the preferred means of indicating errors; however, an unhandled exception will terminate the program.)

### Exceptions

None



---

## cma\_thread\_exit\_normal

Terminates the current thread when successful completion occurs prematurely.

### Syntax

```
cma_thread_exit_normal(  
    result);
```

Argument	Data Type	Access
result	opaque cma_t_address	read

### C Binding

```
#include <cma.h>  
  
void  
cma_thread_exit_normal (  
    cma_t_address result);
```

### Arguments

#### result

Address value that is copied and returned to the caller of `cma_thread_join`.

### Description

This routine terminates execution of the current thread within an arbitrary routine when successful completion occurs prematurely.

Normally, a thread terminates when the *start\_routine* argument to `cma_thread_create` returns. Call this routine when it is not necessary or convenient to allow the thread's start routine to return normally to its caller.

### Exceptions

None



## cma\_thread\_get\_priority

---

### cma\_thread\_get\_priority

Obtains the current priority of a thread.

#### Syntax

```
cma_thread_get_priority(  
    thread,  
    priority);
```

Argument	Data Type	Access
thread	opaque cma_t_thread	read
priority	opaque cma_t_priority	write

#### C Binding

```
#include <cma.h>  
  
void  
cma_thread_get_priority (  
    cma_t_thread  thread,  
    cma_t_priority *priority);
```

#### Arguments

##### **thread**

Handle of the thread whose priority is obtained.

##### **priority**

Variable that receives the current priority value of the thread specified in *thread*. Refer to the description of `cma_thread_set_priority` for valid values.

#### Description

This routine obtains the current priority of a thread. The current priority can be different from the initial priority of the thread if the `cma_thread_set_priority` routine has been called, or if the thread's scheduling policy dynamically modifies thread priorities.

The exact effect of different priority values is dependent upon the scheduling policy assigned to the thread.

#### Exceptions

`cma_e_existence`  
`cma_e_use_error`



## cma\_thread\_get\_sched

Obtains the current scheduling policy of a thread.

### Syntax

```
cma_thread_get_sched(
    thread,
    policy);
```

Argument	Data Type	Access
thread	opaque cma_t_thread	read
policy	opaque cma_t_sched_policy	write

### C Binding

```
#include <cma.h>

void
cma_thread_get_sched (
    cma_t_thread *thread,
    cma_t_sched_policy *policy);
```

### Arguments

#### thread

Handle of the thread whose scheduling policy is obtained.

#### policy

Variable that receives the current scheduling policy value of the thread specified in *thread*. Refer to the description of `cma_thread_set_sched` for valid values.

### Description

This routine obtains the current scheduling policy of a thread. The current scheduling policy of a thread can be different from the initial scheduling policy if the `cma_thread_set_sched` routine has been called.

### Exceptions

```
cma_e_existence
cma_e_use_error
```



---

**cma\_thread\_get\_self**

Obtains the handle of the current thread.

**Syntax**

```
cma_thread_get_self(  
    thread);
```

Argument	Data Type	Access
thread	opaque cma_t_thread	write

**C Binding**

```
#include <cma.h>  
  
void  
cma_thread_get_self (  
    cma_t_thread *thread);
```

**Arguments**

**thread**  
Variable that receives the handle of the current thread.

**Description**

This routine allows a thread to obtain its own handle. This value becomes meaningless when the thread object has been deleted—that is, when the thread has terminated its execution and `cma_thread_detach` has been called.

**Exceptions**

None



## cma\_thread\_join

Causes the calling thread to wait for the termination of a specified thread.

### Syntax

```
cma_thread_join(
    thread,
    exit_status,
    result);
```

Argument	Data Type	Access
thread	opaque cma_t_thread	read
exit_status	opaque cma_t_exit_status	write
result	pointer	write

### C Binding

```
#include <cma.h>

void
cma_thread_join (
    cma_t_thread *thread,
    cma_t_exit_status *exit_status,
    cma_t_address *result);
```

### Arguments

#### thread

Handle of the thread whose termination is awaited by the caller of this routine.

#### exit\_status

Variable that receives a value indicating whether the thread specified by *thread* successfully terminated. Valid values are as follows:

Value	Description
cma_c_term_normal	Normal termination
cma_c_term_error	Error termination (result of calling cma_thread_exit_error() )
cma_c_term_alert	Alert termination

#### result

Address value that is optionally returned by the *start\_routine* of the thread specified by the *thread* argument in its call to cma\_thread\_create.



## cma\_thread\_join

### Description

This routine causes the calling thread to wait for the termination of a specified thread. A call to this routine returns after the specified thread has terminated.

The value returned as the *exit\_status* argument indicates whether the thread terminated normally, because of an error, or because of an alert.

The value returned as the *result* argument is the address that the specified thread generates as its result. The thread's result is normally returned as the value of the *start\_routine* argument in its call to *cma\_thread\_create*. In order for the *result* argument to be valid, the following must occur:

- The *exit\_status* argument must have the value *cma\_c\_term\_normal*.
- The *start\_routine* function for the specified thread must return a value. (Returning a value is optional for the start routine.)

Any number of threads can call this routine. All threads are awakened when the specified thread terminates. If the thread is already terminated, this routine returns immediately.

If the current thread calls this routine, a deadlock results if it is detected by the implementation.

The results of this routine are unpredictable if the value for *thread* refers to *cma\_c\_null* or a thread that has been detached.

This routine is alertable.

### Exceptions

*cma\_e\_alerted*  
*cma\_e\_existence*  
*cma\_e\_use\_error*



---

## cma\_thread\_set\_priority

Changes the current priority of a thread.

### Syntax

```
cma_thread_set_priority(  
    thread,  
    priority);
```

Argument	Data Type	Access
thread	opaque cma_t_thread	read
priority	opaque cma_t_priority	read

### C Binding

```
#include <cma.h>  
  
void  
cma_thread_set_priority (  
    cma_t_thread *thread,  
    cma_t_priority priority);
```

### Arguments

#### thread

Handle of the thread whose priority is changed.

#### priority

New value of the priority attribute. The priority attribute is dependent upon scheduling policy. Valid values are as follows:

Low	Mid	High
cma_c_prio_fifo_min	cma_c_prio_fifo_mid	cma_c_prio_fifo_max
cma_c_prio_rr_min	cma_c_prio_rr_mid	cma_c_prio_rr_max
cma_c_prio_through_min	cma_c_prio_through_mid	cma_c_prio_through_max
cma_c_prio_back_min	cma_c_prio_back_mid	cma_c_prio_back_max

The default priority is cma\_c\_prio\_default\_mid. (This symbol maps to cma\_c\_prio\_through\_mid.)

### Description

This routine changes the current priority of a thread. A thread can change its own priority.

Changing the priority of a thread can cause it to start executing or to be preempted by another thread. The exact effect of setting different priority values depends on the scheduling priority assigned to the thread. The scheduling priority is set by calling the cma\_attr\_set\_priority routine.



## cma\_thread\_set\_priority

An application should specify priority only to express the urgency of executing the thread relative to other threads. Priority should not be used to control mutual exclusion when accessing shared data. With a sufficient number of processors executing, all ready threads, regardless of priority, can be executing simultaneously.

This routine is different from `cma_attr_set_priority` in that `cma_attr_set_priority` sets the priority attribute that is used to establish the priority of a new thread when it is created. However, this routine changes the priority of an existing thread.

### Exceptions

`cma_e_badparam`  
`cma_e_existence`  
`cma_e_unimp`  
`cma_e_use_error`  
`exc_e_nopriv`

Exception	Condition
<code>cma_e_badparam</code>	Invalid priority value
<code>cma_e_existence</code>	Thread does not exist
<code>cma_e_unimp</code>	Operation not implemented
<code>cma_e_use_error</code>	Thread is not in a ready state
<code>exc_e_nopriv</code>	Process does not have sufficient privileges

The `cma_thread_set_priority` routine changes the priority of a thread. The priority value must be a positive integer. The routine returns `CMA_SUCCESS` if the priority is successfully changed. If the routine returns an error, the error code is returned in the `error_code` parameter. The routine is not thread-safe.



---

## cma\_thread\_set\_sched

Changes the current scheduling policy and priority of a thread.

### Syntax

```
cma_thread_set_sched(
    thread,
    policy,
    priority);
```

Argument	Data Type	Access
thread	opaque cma_t_thread	read
policy	opaque cma_t_sched_policy	read
priority	opaque cma_t_priority	read

### C Binding

```
#include <cma.h>

void
cma_thread_set_sched (
    cma_t_thread *thread,
    cma_t_sched_policy policy,
    cma_t_priority priority);
```

### Arguments

#### thread

Handle of the thread whose scheduling policy is changed.

#### policy

New scheduling policy value of the thread specified in *thread*. Valid values are as follows:

```
cma_c_sched_fifo
cma_c_sched_rr
cma_c_sched_throughput
cma_c_sched_background
```

See Section 2.2.3.2 for a description of the scheduling policies.

#### priority

New priority value of the thread specified in *thread*. The priority value is dependent upon scheduling policy. Valid values are as follows:

Low	Mid	High
cma_c_prio_fifo_min	cma_c_prio_fifo_mid	cma_c_prio_fifo_max
cma_c_prio_rr_min	cma_c_prio_rr_mid	cma_c_prio_rr_max
cma_c_prio_through_min	cma_c_prio_through_mid	cma_c_prio_through_max



## cma\_thread\_set\_sched

Low	Mid	High
cma_c_prio_back_min	cma_c_prio_back_mid	cma_c_prio_back_max

The default priority is cma\_c\_prio\_default\_mid. (This symbol maps to cma\_c\_prio\_through\_mid.)

### Description

This routine changes the current scheduling policy and priority of a thread. You can call this routine to change both the priority and scheduling policy of a thread at the same time. To change only the priority, call the cma\_thread\_set\_priority routine.

A thread can change its own scheduling policy and priority. Changing the scheduling policy or priority, or both, of a thread can cause it to start executing or to be preempted by another thread.

This routine is different from cma\_attr\_set\_priority and cma\_attr\_set\_sched in that those routines set the priority and scheduling policy attributes that are used to establish the priority and scheduling policy of a new thread when it is created. This routine, however, changes the priority and scheduling policy of an existing thread.

### Exceptions

cma\_e\_badparam  
cma\_e\_existence  
cma\_e\_unimp  
cma\_e\_use\_error  
exc\_e\_nopriv



## cma\_time\_get\_expiration

Obtains a *cma\_t\_date\_time* value representing a desired expiration time.

### Syntax

```
cma_time_get_expiration(
    expiration,
    interval);
```

Argument	Data Type	Access
expiration	opaque <i>cma_t_date_time</i>	write
interval	single precision floating point	read

### C Binding

```
#include <cma.h>

void
cma_time_get_expiration (
    cma_t_date_time *expiration,
    cma_t_interval interval);
```

### Arguments

#### expiration

Variable that receives the *cma\_t\_date\_time* value representing the expiration time.

#### interval

Number of seconds to add to the current system time. The result is the time that the timed wait should expire.

### Description

This routine adds a specified interval to the current absolute system time and returns a new absolute time. This new absolute time can be used as the expiration time in a call to *cma\_cond\_timed\_wait*.

### Exceptions

```
cma_e_badparam
cma_e_existence
cma_e_use_error
```



## **cma\_unlock\_global**

---

## **cma\_unlock\_global**

Unlocks a global mutex.

### **Syntax**

```
cma_unlock_global( );
```

### **C Binding**

```
#include <cma.h>
```

```
void
```

```
cma_unlock_global(void);
```

### **Arguments**

None

### **Description**

This service unlocks the global mutex when each call to `cma_lock_global` has been matched by a call to this routine. For example, if you called `cma_lock_global` three times, `cma_unlock_global` unlocks the global mutex when you call it the third time.

If no threads are waiting for the global mutex, it becomes unlocked with no current owner. If one or more threads are waiting to lock the global mutex, one thread is unblocked and tries to acquire the global lock again. The scheduling policy is used to determine which thread acquires the global mutex.

The results of calling this routine are unpredictable if the global mutex is already unlocked. The results of calling this service are also unpredictable if the global mutex is owned by a thread other than the calling thread.

Do not call this routine from any software interrupt handler.

### **Exceptions**

`cma_e_existence`

`cma_e_use_error`



---

## cma\_yield

Notifies the scheduler that the current thread is willing to release its processor to other threads of the same priority.

### Syntax

```
cma_yield( );
```

### C Binding

```
#include <cma.h>

void
cma_yield (void);
```

### Arguments

None

### Description

This routine notifies the thread scheduler that the current thread is willing to release its processor to other threads of equivalent or greater scheduling precedence. (A thread generally will release its processor to a thread of a greater scheduling precedence without calling this routine.) If no other threads of equivalent or greater scheduling precedence are ready to execute, the thread continues.

This routine can allow knowledge of the details of an application to be used to improve its performance. If a thread does not call `cma_yield`, other threads may be given the opportunity to run at arbitrary points (possibly even when the interrupted thread holds a required resource). By making strategic calls to `cma_yield`, other threads can be given the opportunity to run when the resources are free. This improves performance by reducing contention for the resource.

As a general guideline, consider calling this routine after a thread has released a resource (such as a mutex) that is heavily contended for by other threads. This can be especially important if the program is running on a uniprocessor machine, or if the thread acquires and releases the resource inside a tight loop.

Use this routine carefully and sparingly, because misuse can cause unnecessary context switching which will increase overhead and actually degrade performance. For example, it is counter-productive for a thread to yield while it holds a resource that the threads to which it is yielding will need. Likewise, it is pointless to yield unless there is likely to be another thread that is ready to run.

### Exceptions

None



## Chris Yild

Chris Yild is a senior software engineer at Google. He has been working at Google for over 10 years and has a passion for building scalable systems.

## Syntax

Chris Yild

## Chris Yild

Chris Yild

Chris Yild

Chris Yild

## Chris Yild

Chris Yild

## Chris Yild

Chris Yild is a senior software engineer at Google. He has been working at Google for over 10 years and has a passion for building scalable systems. Chris Yild is a senior software engineer at Google. He has been working at Google for over 10 years and has a passion for building scalable systems.

Chris Yild is a senior software engineer at Google. He has been working at Google for over 10 years and has a passion for building scalable systems. Chris Yild is a senior software engineer at Google. He has been working at Google for over 10 years and has a passion for building scalable systems.

Chris Yild is a senior software engineer at Google. He has been working at Google for over 10 years and has a passion for building scalable systems. Chris Yild is a senior software engineer at Google. He has been working at Google for over 10 years and has a passion for building scalable systems.

Chris Yild is a senior software engineer at Google. He has been working at Google for over 10 years and has a passion for building scalable systems. Chris Yild is a senior software engineer at Google. He has been working at Google for over 10 years and has a passion for building scalable systems.

## Chris Yild

Chris Yild



# F

---

## DECthreads Library Routines

*For OpenVMS and Digital UNIX Systems Only*

Appendix F provides detailed descriptions of the DECthreads library routines, which make up part of the Digital proprietary interface to DECthreads. These routines specifically allow you to create and control higher-level objects.

To indicate errors, the DECthreads library routines raise exceptions. See Section E.4 for exception descriptions.

DECthreads library routines are callable from programs running on OpenVMS and Digital UNIX systems only.

---

### Note

These routines will no longer be enhanced or documented in future releases.

---



---

## cma\_lib\_attr\_create

Creates a library attributes object.

### Syntax

```
cma_lib_attr_create (new_attr, attr)
```

Argument	Data Type	Access
new_attr	opaque cma_lib_t_attr	write
attr	opaque cma_lib_t_attr	read

### C Binding

```
void  
cma_lib_attr_create (  
    cma_lib_t_attr *new_attr,  
    cma_lib_t_attr *attr);
```

### Arguments

#### **new\_attr**

Variable that receives a handle for the new attributes object.

#### **attr**

Handle of the attributes object used to control attributes of the new attributes object. If you specify `cma_c_null` for the *attr* argument, default attributes are used.

### Description

This routine creates an attributes object that is used to specify the attributes of objects when they are created in routines with the `cma_lib_` prefix.

The `queuesize` attribute is the only currently defined DECthreads library routines attribute. Use the `cma_lib_attr_set_queuesize` routine to change the `queuesize` attribute.

Delete an attributes object by calling the `cma_lib_attr_delete` routine when it is no longer needed to create objects.

### Exceptions

```
cma_e_existence  
cma_e_use_error
```



## cma\_lib\_attr\_delete

Deletes a library attributes object.

### Syntax

cma\_lib\_attr\_delete (attr)

Argument	Data Type	Access
attr	opaque cma_lib_t_attr	read, write

### C Binding

```
void
cma_lib_attr_delete (
cma_lib_t_attr *attr);
```

### Arguments

**attr**  
Handle of the attributes object deleted.

### Description

This routine deletes a library attributes object. The attributes object is marked for deletion, and the *attr* argument is set to the value *cma\_c\_null*. Specifying *cma\_c\_null* for the *attr* argument is legal and has no effect. Objects that were created using this attributes object are not affected by the deletion of the attributes object.

The results of calling this routine are unpredictable if the handle specified by the *attr* argument refers to an attributes object that does not exist (unless it is *cma\_c\_null*.)

### Exceptions

*cma\_e\_existence*  
*cma\_e\_use\_error*



---

## cma\_lib\_attr\_get\_queuesize

Obtains the maximum number of elements available on an atomic queue attribute that is used when a queue is created.

### Syntax

cma\_lib\_attr\_get\_queuesize (attr, queuesize)

Argument	Data Type	Access
attr	opaque cma_lib_t_attr	read
queuesize	opaque cma_t_natural	write

### C Binding

```
void  
cma_lib_attr_get_queuesize (  
    cma_lib_t_attr *attr,  
    cma_t_natural *queuesize);
```

### Arguments

**attr**

Handle of the library attributes object whose queuesize is obtained. This value is returned by cma\_lib\_attr\_create.

**queuesize**

Variable that receives the current value of the queuesize attribute.

### Description

This routine obtains the queuesize attribute that is used when a queue is created. The queuesize attribute specifies the maximum number of elements allowed on a queue. The default queuesize attribute is 128.

### Exceptions

cma\_e\_existence  
cma\_e\_use\_error



## cma\_lib\_attr\_set\_queue size

Specifies the attribute for the maximum number of elements allowed on an atomic queue that is used when a queue is created.

### Syntax

cma\_lib\_attr\_set\_queue size (attr, queue size)

Argument	Data Type	Access
attr	opaque cma_lib_t_attr	read
queue size	opaque cma_t_natural	read

### C Binding

```
void
cma_lib_attr_set_queue size (
    cma_lib_t_attr *attr,
    cma_t_natural queue size);
```

### Arguments

#### attr

Handle of the library attributes object to be modified. This value is returned by cma\_lib\_attr\_create.

#### queue size

New value for the queue size attribute. The default value is 128.

### Description

This routine sets the queue size attribute that is used when a queue is created. The queue size attribute specifies the maximum number of elements allowed on a queue.

A queue contains a fixed number of available queue items. Call this routine if you want to increase or decrease the queue size of new queues when they are created.

### Exceptions

```
cma_e_badparam
cma_e_existence
cma_e_use_error
```



## cma\_lib\_queue\_create

Creates an atomic queue.

### Syntax

```
cma_lib_queue_create (new_queue, attr )
```

Argument	Data Type	Access
new_queue	opaque cma_lib_t_queue	write
attr	opaque cma_lib_t_attr	read

### C Binding

```
void  
cma_lib_queue_create (  
    cma_lib_t_queue *new_queue,  
    cma_lib_t_attr *attr);
```

### Arguments

**new\_queue**

Variable that receives the handle of the atomic queue created.

**attr**

Library attributes object used when creating the queue.

### Description

This routine creates an atomic queue. Unlike mutexes and condition variables, an atomic queue implements interthread communication that is not explicitly dependent on synchronization. A queue can communicate information among threads or within a single thread (for example, between an interrupt routine and the normal thread code).

The primary characteristic of an atomic queue is that any operation on the queue is guaranteed to complete before any other operation on that same queue can begin.

The created queue is not affected by termination of the thread that created it. It remains valid until explicitly deleted by `cma_lib_queue_delete`.

### Exceptions

`cma_e_existence`  
`cma_e_use_error`



## cma\_lib\_queue\_delete

Deletes an atomic queue.

### Syntax

```
cma_lib_queue_delete (queue)
```

Argument	Data Type	Access
queue	opaque cma_lib_t_queue	read, write

### C Binding

```
void
cma_lib_queue_delete (
cma_lib_t_queue *queue);
```

### Arguments

**queue**  
Handle of the queue to be deleted.

### Description

This routine deletes the specified atomic queue. After deletion, the handle is set to cma\_c\_null. A queue remains valid until explicitly deleted by cma\_lib\_queue\_delete.

A queue must be empty for it to be deleted. If the queue is not empty when you call this routine, the exception cma\_e\_in\_use is raised.

### Exceptions

cma\_e\_existence  
cma\_e\_in\_use  
cma\_e\_use\_error



---

## cma\_lib\_queue\_dequeue

Removes the first element from an atomic queue.

### Syntax

cma\_lib\_queue\_dequeue (queue, element)

Argument	Data Type	Access
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	write

### C Binding

```
void  
cma_lib_queue_dequeue (  
    cma_lib_t_queue *queue,  
    cma_lib_t_address *element);
```

### Arguments

**queue**

Handle of the queue from which the element is removed.

**element**

Variable that receives the address of the removed queue element.

### Description

This routine removes the first element from an atomic queue. If the queue is empty, the calling thread is blocked until an element is inserted into the queue. When the element is enqueued the calling thread resumes, the new element is removed, and this routine returns.

Call `cma_lib_queue_try_dequeue` to remove an element from a queue and return a status code (instead of blocking) if the queue is empty.

### Exceptions

cma\_e\_existence  
cma\_e\_use\_error



## cma\_lib\_queue\_enqueue

Inserts an element at the end of an atomic queue.

### Syntax

cma\_lib\_queue\_enqueue (queue, element)

Argument	Data Type	Access
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	read

### C Binding

```
void
cma_lib_queue_enqueue (
cma_lib_t_queue *queue,
cma_lib_t_address element);
```

### Arguments

**queue**  
Handle of the queue to which the element is inserted.

**element**  
Address of the queue element inserted.

### Description

This routine inserts an element at the end of a queue. If the queue is full, the calling thread is blocked until an element is removed from the queue. When the element is dequeued the calling thread resumes, the new element is inserted, and this routine returns.

Call cma\_lib\_queue\_try\_enqueue to insert an element into a queue and return a status code (instead of blocking) if the queue is full.

### Exceptions

cma\_e\_existence  
cma\_e\_use\_error



---

## cma\_lib\_queue\_requeue

Inserts an element at the front of an atomic queue.

### Syntax

cma\_lib\_queue\_requeue (queue, element)

Argument	Data Type	Access
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	read

### C Binding

```
void  
cma_lib_queue_requeue (  
    cma_lib_t_queue *queue,  
    cma_lib_t_address element);
```

### Arguments

**queue**

Handle of the queue to which the element is inserted.

**element**

Address of the queue element inserted.

### Description

This routine inserts an element at the front of a queue. If the queue is full, the calling thread is blocked until an element is removed from the queue. When the element is dequeued the calling thread resumes, the new element is inserted, and this routine returns.

Call `cma_lib_queue_try_requeue` to insert an element at the front of a queue and return a status code (instead of blocking) if the queue is full.

This routine allows you to replace an element that was erroneously removed from a queue. For example, a queue might hold information of various types. In that case a thread can remove the oldest (first) element of the queue, check its type, and requeue the element if it is not the desired type (rather than enqueueing it, which would place the element at the end of the queue).

### Exceptions

cma\_e\_existence  
cma\_e\_use\_error



---

**cma\_lib\_queue\_try\_dequeue**

Removes the first element from an atomic queue.

**Syntax**

```
status = cma_lib_queue_try_dequeue (queue, element)
```

Argument	Data Type	Access
status	Boolean cma_t_boolean	write
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	write

**C Binding**

```

cma_t_boolean
cma_lib_queue_dequeue (
cma_lib_t_queue *queue,
cma_lib_t_address *element);

```

**Arguments****status**

Boolean value that specifies whether the element was dequeued.

**queue**

Handle of the queue from which the element is removed.

**element**

Variable that receives the address of the removed queue element.

**Description**

This routine removes the first element from an atomic queue. If the queue is empty, the calling thread does not wait for an element to be enqueued. Instead, the routine returns with the status `cma_c_false`. If the queue is not empty, the first element value is returned to *element*, and the routine returns with the status `cma_c_true`.

Call `cma_lib_queue_dequeue` if you want to remove an element from a queue but cause the calling thread to block if the queue is empty.

**Exceptions**

```

cma_e_existence
cma_e_use_error

```



---

## cma\_lib\_queue\_try\_enqueue

Inserts an element at the end of an atomic queue.

### Syntax

```
status = cma_lib_queue_try_enqueue (queue, element)
```

Argument	Data Type	Access
status	Boolean cma_t_boolean	write
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	read

### C Binding

```
cma_t_boolean  
cma_lib_queue_try_enqueue (  
cma_lib_t_queue *queue,  
cma_lib_t_address element);
```

### Arguments

**status**

Boolean value that specifies whether the element was enqueued.

**queue**

Handle of the queue to which the element is inserted.

**element**

Address of the queue element inserted.

### Description

This routine inserts an element at the end of a queue. If the queue is full, the calling thread does not wait for an element to be removed. Instead, the routine returns with the status `cma_c_false`. If the queue is not full, the element is inserted at the end of the queue and the routine returns with the status `cma_c_true`.

Call `cma_lib_queue_enqueue` if you want to insert an element into a queue but cause the calling thread to block if the queue is full. Call `cma_lib_queue_try_enqueue_int` if you want to insert an element into a queue from interrupt level.

### Exceptions

`cma_e_existence`  
`cma_e_use_error`



## cma\_lib\_queue\_try\_enqueue\_int

Inserts an element at the end of an atomic queue from interrupt level.

### Syntax

```
status = cma_lib_queue_try_enqueue_int (queue, element)
```

Argument	Data Type	Access
status	Boolean cma_t_boolean	write
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	read

### C Binding

```
cma_t_boolean  
cma_lib_queue_try_enqueue_int (  
cma_lib_t_queue *queue,  
cma_lib_t_address element);
```

### Arguments

**status**  
Boolean value that specifies whether the element was enqueued.

**queue**  
Handle of the queue to which the element is inserted.

**element**  
Address of the queue element inserted.

### Description

This routine inserts an element at the end of a queue from interrupt level. If the queue is full, the calling thread does not wait for an element to be removed. Instead, the routine returns with the status `cma_c_false`. If the queue is not full, the element is inserted at the end of the queue and the routine returns with the status `cma_c_true`.

#### Note

This routine allows you to add elements to a queue from a software interrupt handler. Do not call this routine from noninterrupt code. If you want to add elements to a queue from the normal noninterrupt level without blocking, use `cma_lib_queue_try_enqueue`.

### Exceptions

`cma_e_existence`  
`cma_e_use_error`



## cma\_lib\_queue\_try\_requeue

Inserts an element at the front of an atomic queue.

### Syntax

```
status = cma_lib_queue_try_requeue (queue, element)
```

Argument	Data Type	Access
status	Boolean cma_t_boolean	write
queue	opaque cma_lib_t_queue	read
element	opaque cma_t_address	read

### C Binding

```
cma_t_boolean  
cma_lib_queue_requeue (  
cma_lib_t_queue *queue,  
cma_lib_t_address element);
```

### Arguments

#### status

Boolean value that specifies whether the element was requeued.

#### queue

Handle of the queue to which the element is inserted.

#### element

Address of the queue element inserted.

### Description

This routine inserts an element at the front of a queue. If the queue is full, the calling thread does not wait until an element is removed from the queue. Instead, the routine returns with the status `cma_c_false`. If the queue is not full, the element is added to the front of the queue and the routine returns with the status `cma_c_true`.

Call `cma_lib_queue_requeue` if you want to insert an element into a queue but cause the calling thread to block if the queue is full.

This routine allows you to replace an element that was erroneously removed from a queue. For example, a queue might hold information of various types. In that case a thread can remove the oldest (first) element of the queue, check its type, and requeue the element if it is not the desired type (rather than enqueueing it, which would place the element at the end of the queue).

### Exceptions

```
cma_e_existence  
cma_e_use_error
```



---

## POSIX 1003.4a (Draft 4) pthread Routines

This appendix provides migration information and detailed descriptions of the POSIX 1003.4a (Draft 4) pthread routines that make up the older interface to DECthreads to support your code upgrade. (POSIX 1003.4a was renamed to POSIX 1003.1c at a later POSIX draft standard.)

---

### Note

---

The pthread routines described in this appendix are based on POSIX 1003.4a Draft 4. For the latest version (POSIX 1003.1c) of DECthreads routines, see Part II. Users should be aware that applications consistent with the P1003.4a Draft 4 routines described in this appendix require significant modification to upgrade to the 1003.1c standard interface described in Part II, which reflects the latest conforming POSIX standard of DECthreads.

---

To indicate errors, the POSIX 1003.1c standard pthread routines return status values by default. (See Chapter 5 for a method of allowing the pthread routines to raise exceptions.)

Routine names ending with the `_np` suffix denote that the routine is not portable—the routine might not be available in implementations of POSIX thread application interfaces other than DECthreads. You need not prototype the pthread routines if you include the POSIX 1003.1c `pthread.h`.

### G.1 Migrating from a 1003.4a Interface to POSIX 1003.1c

The new DECthreads POSIX 1003.1c interface is significantly different than the original 1003.4a DECthreads implementation. This section describes the major changes between the interfaces. For further routine information, refer to Part II of this guide.

#### G.1.1 Error Status and Function Returns

The new DECthreads POSIX 1003.1c interface does not use *errno*. (Note that DECthreads still provides a thread-specific *errno* cell for use by libraries and application code, but the 1003.1c interface does not write to this cell.) If an error condition occurs, a pthread routine returns an integer value indicating the type of error. For example, a call to the Draft 4 implementation of `pthread_cond_destroy` that returned a `-1` and set *errno* to `EBUSY`, now returns `EBUSY` as the routine return value in the current implementation. On successful completion, most pthread routines return a zero.



## G.1.2 Replaced or Renamed Routines

Many routines have been replaced or renamed as follows:

Draft 4 Routines	Replaced by POSIX 1003.1c Routines
pthread_attr_create	pthread_attr_init
pthread_attr_delete	pthread_attr_destroy
pthread_attr_set/getdetach_np	pthread_attr_set/getdetachstate
pthread_attr_set/getprio	pthread_attr_set/getschedparam
pthread_attr_set/getsched	pthread_attr_set/getschedpolicy
pthread_condattr_create	pthread_condattr_init
pthread_condattr_delete	pthread_condattr_destroy
pthread_keycreate	pthread_key_create
pthread_mutexattr_create	pthread_mutexattr_init
pthread_mutexattr_delete	pthread_mutexattr_destroy
pthread_mutexattr_get/setkind_np	pthread_mutexattr_get/settype_np
pthread_setasynccancel	pthread_setcanceltype
pthread_setcancel	pthread_setcancelstate
pthread_set/getprio	pthread_set/getschedparam
pthread_set/getscheduler	pthread_set/getschedparam
pthread_yield	sched_yield

## G.1.3 Routines with No Changes to Syntax

Except for their return value, the following routines have no changes to their syntax:

- pthread\_attr\_setinheritsched
- pthread\_cancel
- pthread\_cond\_broadcast
- pthread\_cond\_destroy
- pthread\_cond\_signal
- pthread\_cond\_signal\_int\_np
- pthread\_cond\_timedwait
- pthread\_cond\_wait
- pthread\_delay\_np
- pthread\_equal
- pthread\_exit
- pthread\_get\_expiration\_np
- pthread\_join (now detaches the thread)
- pthread\_mutex\_destroy
- pthread\_mutex\_lock
- pthread\_mutex\_trylock
- pthread\_mutex\_unlock
- pthread\_once

The following routines have no changes:

- pthread\_self
- pthread\_testcancel



## G.1.4 Routines with Prototype or Syntax Changes

The following routines have changes to their argument syntax with the new DECthreads implementation:

Old Syntax	New Syntax
unsigned long <b>pthread_attr_getguardsize_np</b> (pthread_attr_t attr)	int <b>pthread_attr_getguardsize_np</b> (const pthread_attr_t *attr, size_t *guardsize)
int <b>pthread_attr_getinheritsched</b> (pthread_attr_t attr)	int <b>pthread_attr_getinheritsched</b> (const pthread_attr_t *attr, int *inheritsched)
unsigned long <b>pthread_attr_getstacksize</b> (pthread_attr_t attr)	int <b>pthread_attr_getstacksize</b> (const pthread_attr_t *attr, size_t *stacksize)
unsigned long <b>pthread_attr_setguardsize_np</b> (pthread_attr_t *attr, long guardsize)	int <b>pthread_attr_setguardsize_np</b> (pthread_attr_t *attr, size_t guardsize)
unsigned long <b>pthread_attr_setstacksize</b> (pthread_attr_t *attr, long stacksize)	int <b>pthread_attr_setstacksize</b> (const pthread_attr_t *attr, size_t stacksize)
int <b>pthread_cleanup_pop</b> (int execute)	void <b>pthread_cleanup_pop</b> (int execute)
int <b>pthread_cond_init</b> (pthread_cond_t *cond, pthread_condattr_t attr)	int <b>pthread_cond_init</b> (pthread_cond_t *cond, pthread_condattr_t *attr)
int <b>pthread_create</b> (pthread_t *thread, pthread_attr_t attr, pthread_startroutine_t start_routine, pthread_addr_t arg)	int <b>pthread_create</b> (pthread_t *thread, const pthread_attr_t *attr, void* (*start_routine)(void*), void *arg)
int <b>pthread_detach</b> (pthread_t *thread)	int <b>pthread_detach</b> (pthread_t thread)
int <b>pthread_getspecific</b> (pthread_key_t key, void **value)	void * <b>pthread_getspecific</b> (pthread_key_t key)
void <b>pthread_lock_global_np</b> ()	int <b>pthread_lock_global_np</b> (void)
void <b>pthread_unlock_global_np</b> ()	int <b>pthread_unlock_global_np</b> (void)
int <b>pthread_mutex_init</b> (pthread_mutex_t *mutex, pthread_mutexattr_t attr)	int <b>pthread_mutex_init</b> (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)

The following routines no longer support previous nonstandard DECthreads datatypes that were documented in the previous release. If your original code used the standard (real) datatypes, then this list may not impact your code.

New Standard Datatype Syntax	Nonstandard Datatype Syntax
void <b>pthread_cleanup_push</b> (void (*routine)(void *), void *arg)	int <b>pthread_cleanup_push</b> (pthread_cleanup_t *routine, pthread_addr_t arg)
int <b>pthread_create</b> (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)	int <b>pthread_create</b> (pthread_t *thread, pthread_attr_t attr, pthread_startroutine_t start_routine, pthread_addr_t arg)
int <b>pthread_exit</b> (void *value_ptr)	int <b>pthread_exit</b> (pthread_addr_t status)
void * <b>pthread_getspecific</b> (pthread_key_t key)	int <b>pthread_getspecific</b> (pthread_key_t key, pthread_addr_t *value)



---

**New Standard Datatype Syntax**

```
int pthread_join(  
    pthread_t thread, void **value_ptr)  
int pthread_once(  
    pthread_once_t *once_control, void (*init_  
    routine)(void))  
int pthread_setspecific(  
    pthread_key_t key, const void *value)
```

---

**Nonstandard Datatype Syntax**

```
int pthread_join(  
    pthread_t thread, pthread_addr_t *status)  
int pthread_once(  
    pthread_once_t *once_block, pthread_  
    initroutine_t init_routine)  
int pthread_setspecific(  
    pthread_key_t key, pthread_addr_t value)
```

---

### G.1.5 New Routines

The following are new POSIX 1003.1c pthread routines that did not exist at the time of the 1003.4a Draft 4 implementation:

- pthread\_atfork (UNIX only)
- pthread\_attr\_getdetachstate
- pthread\_attr\_setdetachstate
- pthread\_key\_delete
- pthread\_kill (UNIX only)
- pthread\_sigmask (UNIX only)



## pthread\_attr\_create

Creates a thread attributes object.

### Syntax

```
pthread_attr_create(
    attr);
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

### C Binding

```
int
pthread_attr_create (
    pthread_attr_t *attr);
```

### Arguments

**attr**  
Thread attributes object created.

### Description

This routine creates a thread attributes object that is used to specify the attributes of threads when they are created. The attributes object created by this routine is only used in calls to pthread\_create.

The individual attributes (internal fields) of the attributes object are set to default values. (The default values of each attribute are discussed in the descriptions of the following routines.) Use the following routines to change the individual attributes:

```
pthread_attr_setinheritsched
pthread_attr_setprio
pthread_attr_setsched
pthread_attr_setstacksize
```

When an attributes object is used to create a thread, the values of the individual attributes determine the characteristics of the new object. Attributes objects perform similar to additional arguments to object creation. Changing individual attributes does not affect any objects that were previously created using the attributes object.

When you set the scheduling policy or priority, or both, in an attributes object, you must disable scheduling inheritance before the scheduling attributes are used.



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_attr\_create

#### Return Values

If an error condition occurs, this routine returns `-1` and sets `errno` to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[ENOMEM]	Insufficient memory exists to create the thread attributes object.
-1	[EINVAL]	The value specified by <i>attr</i> is invalid.



## pthread\_attr\_delete

Deletes a thread attributes object.

### Syntax

```
pthread_attr_delete(
    attr );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

### C Binding

```
int
pthread_attr_delete (
    pthread_attr_t *attr);
```

### Arguments

**attr**  
Thread attributes object deleted.

### Description

This routine deletes a thread attributes object. This routine gives permission to reclaim storage for the thread attributes object. Threads that were created using this thread attributes object are not affected by the deletion of the thread attributes object.

The results of calling this routine are unpredictable if the value specified by the *attr* argument refers to a thread attributes object that does not exist.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[ENOMEM]	Insufficient memory exists to create the thread attributes object.
-1	[EINVAL]	The value specified by <i>attr</i> is invalid.
-1	[ESRCH]	The value specified by <i>attr</i> does not refer to an existing thread attributes object.



## pthread\_attr\_getdetach\_np

Obtains the detachstate attribute of thread creation.

### Syntax

```
pthread_attr_getdetach_np( pthread_attr_t *attr);
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

### C Binding

```
int  
pthread_attr_getdetach_np (  
pthread_attr_t *attr);
```

### Arguments

**attr**  
Thread attributes object whose detachstate attribute is obtained.

### Description

This routine obtains the detachstate attribute of thread creation. This attribute specifies whether threads created using the specified thread attributes object are created in detached state.

The default value of the detachstate attribute is `PTHREAD_CREATE_JOINABLE`.

See the `pthread_attr_setdetach_np` description for information about the detachstate attribute.

### Return Values

On successful completion, this routine returns the detachstate attribute value. The value is `PTHREAD_CREATE_JOINABLE` to create threads that are not detached, or `PTHREAD_CREATE_DETACHED` to create threads that are detached.

If an error condition occurs, this routine returns `-1` and sets `errno` to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Detachstate attribute		Successful completion.
-1	[EINVAL]	The value specified by the detachstate attribute is invalid.
-1	[ESRCH]	The value specified by <i>attr</i> does not refer to an existing thread attributes object.



## pthread\_attr\_getguardsize\_np

Obtains the guardsize attribute of the specified thread attributes object.

### Syntax

```
pthread_attr_getguardsize_np(
    attr );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

### C Binding

```
unsigned long
pthread_attr_getguardsize_np (
    pthread_attr_t attr);
```

### Arguments

**attr**

Thread attributes object whose guardsize attribute is obtained.

### Description

This routine obtains the minimum size (in bytes) of the guard area for the stack of a thread that is created using the attributes object specified by the *attr* argument.

A guard area helps to detect stack overflows by preventing memory access beyond the thread's stack. Large guard areas are necessary when threads might allocate large structures on the stack.

### Return Values

On successful completion, this routine returns the guardsize attribute value.

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Guardsize attribute		Successful completion.
-1	[EINVAL]	The value specified by <i>attr</i> is invalid.



## pthread\_attr\_getinheritsched

Obtains the inherit scheduling attribute of the specified thread attributes object.

### Syntax

```
pthread_attr_getinheritsched(  
    attr );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

### C Binding

```
int  
pthread_attr_getinheritsched (  
    pthread_attr_t attr);
```

### Arguments

**attr**

Thread attributes object whose inherit scheduling attribute is obtained.

### Description

This routine obtains the value of the inherit scheduling attribute in the specified thread attributes object. The inherit scheduling attribute specifies whether threads created using the attributes object inherit the scheduling attributes of the creating thread, or use the scheduling attributes stored in the attributes object that is passed to pthread\_create.

The default value of the inherit scheduling attribute is PTHREAD\_INHERIT\_SCHED.

### Return Values

On successful completion, this routine returns the inherit scheduling attribute value.

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Inherit scheduling attribute		Successful completion.
-1	[EINVAL]	The value specified by <i>attr</i> is invalid.
-1	[ESRCH]	The value specified by <i>attr</i> does not refer to an existing thread attributes object.



## pthread\_attr\_getprio

Obtains the scheduling priority attribute of the specified thread attributes object.

### Syntax

```
pthread_attr_getprio(
    attr);
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

### C Binding

```
int
pthread_attr_getprio (
    pthread_attr_t attr);
```

### Arguments

**attr**

Thread attributes object whose priority attribute is obtained.

### Description

This routine obtains the value of the scheduling priority of threads created using the thread attributes object specified by the *attr* argument.

### Return Values

On successful completion, this routine returns the scheduling priority attribute value.

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Scheduling priority attribute		Successful completion.
-1	[EINVAL]	The value specified by <i>attr</i> is invalid.
-1	[ESRCH]	The value specified by <i>attr</i> does not refer to an existing thread attributes object.



## pthread\_attr\_getsched

Obtains the scheduling policy attribute of the specified thread attributes object.

### Syntax

```
pthread_attr_getsched(  
    attr);
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

### C Binding

```
int  
pthread_attr_getsched (  
    pthread_attr_t attr);
```

### Arguments

**attr**  
Thread attributes object whose scheduling policy attribute is obtained.

### Description

This routine obtains the scheduling policy of threads created using the thread attributes object specified by the *attr* argument. The default value of the scheduling attribute is `SCHED_OTHER`.

### Return Values

On successful completion, this routine returns the value of the scheduling policy attribute.

If an error condition occurs, this routine returns `-1` and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Scheduling policy attribute		Successful completion.
<code>-1</code>	<code>[EINVAL]</code>	The value specified by <i>attr</i> is invalid.
<code>-1</code>	<code>[ESRCH]</code>	The value specified by <i>attr</i> does not refer to an existing thread attributes object.



## pthread\_attr\_getstacksize

Obtains the stacksize attribute of the specified thread attributes object.

### Syntax

```
pthread_attr_getstacksize(
    attr);
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read

### C Binding

```
unsigned long
pthread_attr_getstacksize (
pthread_attr_t attr);
```

### Arguments

**attr**

Thread attributes object whose stacksize attribute is obtained.

### Description

This routine obtains the minimum size (in bytes) of the stack for a thread created using the thread attributes object specified by the *attr* argument.

### Return Values

On successful completion, this routine returns the stacksize attribute value.

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Stacksize attribute		Successful completion.
-1	[EINVAL]	The value specified by <i>attr</i> is invalid.



## pthread\_attr\_setdetach\_np

Changes the detachstate attribute of thread creation.

### Syntax

```
pthread_attr_setdetach_np( pthread_attr_t *attr, int detachstate);
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
detachstate	integer	read

### C Binding

```
int  
pthread_attr_setdetach_np(  
pthread_attr_t *attr,  
int detachstate);
```

### Arguments

**attr**  
Thread attributes object to be modified.

**detachstate**  
New value for the detachstate attribute. Valid values are as follows:

PTHREAD_CREATE_JOINABLE	This is the default value. Threads are created in “undetached” state.
PTHREAD_CREATE_DETACHED	The created thread is detached immediately, before it begins running.

### Description

This routine changes the detachstate attribute of thread creation. This attribute specifies whether threads created using the specified thread attributes object are created in detached state.

You cannot use the thread handle (the value of type `thread_t` that is returned by `pthread_create`) for a detached thread. This means that you cannot cancel the thread with `pthread_cancel`. You also cannot use `pthread_join` to wait for the thread to complete or to retrieve the thread’s return status.

When a thread that has not been detached completes execution, DECthreads will retain the state of that thread to allow another thread to join with it. If the thread is detached before it completes, DECthreads is free to immediately reclaim the thread’s storage and resources. Failing to detach threads that have completed execution can result in wasting resources, so threads should be detached as soon as the program is done with them. If there is no need to use the thread’s handle after creation, create the thread initially detached.



## Return Values

If an error condition occurs, this routine returns `-1` and sets `errno` to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by the detachstate attribute is invalid.
-1	[ESRCH]	The value specified by <i>attr</i> does not refer to an existing thread attributes object.



## pthread\_attr\_setguardsize\_np

Changes the guardsize attribute of thread creation.

### Syntax

```
pthread_attr_setguardsize_np(  
    attr,  
    guardsize );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
guardsize	longword	read

### C Binding

```
int  
pthread_attr_setguardsize_np (  
    pthread_attr_t *attr,  
    long guardsize);
```

### Arguments

**attr**  
Threads attributes object modified.

**guardsize**  
New value for the guardsize attribute. The *guardsize* argument specifies the minimum size (in bytes) of the guard area for the stack of a thread.

### Description

This routine sets the minimum size (in bytes) of the guard area for the stack of a thread that is created using the attributes object specified by the *attr* argument.

A guard area helps to detect stack overflows by preventing memory access beyond the thread's stack. Large guard areas are necessary when threads might allocate large structures on the stack.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>attr</i> is invalid.



## pthread\_attr\_setinheritsched

Changes the inherit scheduling attribute of the specified thread attributes object.

### Syntax

```
pthread_attr_setinheritsched(
    attr,
    inherit);
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
inherit	integer	read

### C Binding

```
int
pthread_attr_setinheritsched (
pthread_attr_t *attr,
int inherit);
```

### Arguments

#### attr

Thread attributes object to be modified.

#### inherit

New value for the inherit scheduling attribute. Valid values are as follows:

PTHREAD\_INHERIT\_ SCHED

This is the default value. The created thread inherits the current priority and scheduling policy of the thread calling pthread\_create.

PTHREAD\_DEFAULT\_ SCHED

The created thread starts execution with the priority and scheduling policy stored in the thread attributes object.

### Description

This routine changes the inherit scheduling attribute of thread creation. The inherit scheduling attribute specifies whether threads created using the specified thread attributes object inherit the scheduling attributes of the creating thread, or use the scheduling attributes stored in the thread attributes object that is passed to pthread\_create.

The first thread in an application that is not created by an explicit call to pthread\_create has a scheduling policy of SCHED\_OTHER. See the pthread\_attr\_setprio and pthread\_attr\_setsched routines for more information on valid priority values and valid scheduling policy values, respectively.

Inheriting scheduling attributes (instead of using the scheduling attributes stored in the attributes object) is useful when a thread is creating several helper threads—threads that are intended to work closely with the creating thread to cooperatively solve the same problem. For example, inherited scheduling



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_attr\_setinheritsched

attributes ensure that helper threads created in a sort routine execute with the same priority as the calling thread.

#### Return Values

If an error condition occurs, this routine returns `-1` and sets `errno` to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>inherit</i> is invalid.
-1	[ESRCH]	The value specified by <i>attr</i> does not refer to an existing thread attributes object.



## pthread\_attr\_setprio

Changes the scheduling priority attribute of thread creation.

### Syntax

```
pthread_attr_setprio(
    attr,
    priority);
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
priority	integer	read

### C Binding

```
int
pthread_attr_setprio (
    pthread_attr_t *attr,
    int priority);
```

### Arguments

#### attr

Thread attributes object modified.

#### priority

New value for the priority attribute. The priority attribute is dependent upon scheduling policy. Valid values fall within one of the following ranges:

Low	High
PRI_FIFO_MIN	PRI_FIFO_MAX
PRI_RR_MIN	PRI_RR_MAX
PRI_OTHER_MIN	PRI_OTHER_MAX
PRI_FG_MIN_NP	PRI_FG_MAX_NP
PRI_BG_MIN_NP	PRI_BG_MAX_NP

The default priority is the midpoint between PRI\_OTHER\_MIN and PRI\_OTHER\_MAX. (Section 2.7 describes how to specify priorities between the minimum and maximum values.)

### Description

This routine sets the execution priority of threads that are created using the attributes object specified by the *attr* argument.

By default, a created thread inherits the priority of the thread calling pthread\_create. To specify a priority using this routine, scheduling inheritance must be disabled at the time the thread is created. Call pthread\_attr\_setinheritsched and specify the value PTHREAD\_DEFAULT\_SCHED for the *inherit* argument before calling pthread\_create.



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_attr\_setprio

An application specifies priority only to express the urgency of executing the thread relative to other threads. Priority is not used to control mutual exclusion when accessing shared data. With a sufficient number of processors executing, all ready threads, regardless of priority, execute simultaneously.

### Return Values

If an error condition occurs, this routine returns `-1` and sets `errno` to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>priority</i> is invalid.
-1	[ERANGE]	One or more arguments supplied have an invalid value.



## pthread\_attr\_setsched

Changes the scheduling policy attribute of thread creation.

### Syntax

```
pthread_attr_setsched(
    attr,
    scheduler );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
scheduler	integer	read

### C Binding

```
int
pthread_attr_setsched (
    pthread_attr_t *attr,
    int scheduler);
```

### Arguments

#### attr

Threads attributes object modified.

#### scheduler

New value for the scheduling policy attribute. (Policies listed on the same line are equivalent.) Valid values are as follows:

```
SCHED_FIFO
SCHED_RR
SCHED_FG_NP or SCHED_OTHER
SCHED_BG_NP
```

See Section 2.2.3.2 for a description of the scheduling policies.

### Description

This routine sets the scheduling policy of a thread that is created using the attributes object specified by the *attr* argument. The default value of the scheduling attribute is `SCHED_OTHER`.

By default, a created thread inherits the priority of the thread calling `pthread_create`. To specify a priority using this routine, scheduling inheritance must be disabled at the time the thread is created. Call `pthread_attr_setinheritsched` and specify the value `PTHREAD_DEFAULT_SCHED` for the *inherit* argument before calling `pthread_create`.



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_attr\_setsched

#### Return Values

If an error condition occurs, this routine returns `-1` and sets `errno` to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>scheduler</i> is invalid.



## pthread\_attr\_setstacksize

Changes the stacksize attribute of thread creation.

### Syntax

```
pthread_attr_setstacksize(
    attr,
    stacksize );
```

Argument	Data Type	Access
attr	opaque pthread_attr_t	read
stacksize	longword	read

### C Binding

```
int
pthread_attr_setstacksize (
    pthread_attr_t *attr,
    long stacksize);
```

### Arguments

**attr**  
Threads attributes object modified.

**stacksize**  
New value for the stacksize attribute. The *stacksize* argument must be greater than or equal to zero. It specifies the minimum size (in bytes) of stack needed for a thread.

### Description

This routine sets the minimum size (in bytes) of the stack needed for a thread created using the attributes object specified by the *attr* argument. Use this routine to adjust the size of the writable area of the stack. The default stacksize value is available by calling pthread\_attr\_getstacksize on a newly created attributes object.

A thread's stack is fixed at the time of thread creation. Only the main or initial thread can dynamically extend its stack.

Most compilers do not check for stack overflow. Ensure that your thread stack is large enough for anything that you call from the thread.



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_attr\_setstacksize

#### Return Values

If an error condition occurs, this routine returns `-1` and sets `errno` to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>attr</i> is invalid.



## pthread\_bind\_to\_cpu\_np

Binds a thread to a particular CPU on a multiprocessor system.

### Syntax

```
pthread_bind_to_cpu_np(
    thread, cpu_mask );
```

Argument	Data Type	Access
thread	opaque pthread_t	read
cpu_mask	unsigned long	read

### C Binding

```
int
pthread_bind_to_cpu_np (
    pthread_t thread,
    unsigned long cpu_mask);
```

### Arguments

#### thread

Thread that will be bound to a CPU.

#### cpu\_mask

Bit mask specifying which CPU the thread is to be bound to. The low-order bit represents the first CPU. Only one bit in this mask may be set.

### Description

This routine binds a thread to a particular processor on a multiprocessor system. Once bound, a thread will only run on the specified CPU until the thread terminates, or the binding is changed.

Specify a *cpu\_mask* of 0 to allow a previously bound thread to execute on any available CPU.

This routine is not available on all platforms. If it is not available, `pthread_bind_to_cpu_np` returns -1 and sets *errno* to ENOSYS.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>thread</i> or <i>cpu_mask</i> is invalid.
-1	[ENOSYS]	The function is not supported on this system.



## pthread\_cancel

Allows a thread to request that it or another thread terminate execution.

### Syntax

```
pthread_cancel(  
    thread );
```

Argument	Data Type	Access
thread	opaque pthread_t	read

### C Binding

```
int  
pthread_cancel (  
    pthread_t thread);
```

### Arguments

**thread**

Thread that receives a cancel request.

### Description

This routine sends a cancel to the specified thread. A cancel is a mechanism by which a calling thread informs the specified thread to terminate as quickly as possible. Issuing a cancel does not guarantee that the canceled thread will receive or handle the cancel. The canceled thread can delay processing the cancel after receiving it. For instance, if a cancel arrives during an important operation, the canceled thread can continue if what it is doing cannot be interrupted at the point where the cancel is requested.

Because of communication delays, the calling thread can only rely on the fact that a cancel will eventually become pending in the designated thread (provided that the thread does not terminate beforehand). Furthermore, the calling thread has no guarantee that a pending cancel will be delivered because delivery is controlled by the designated thread.

Termination processing when a cancel is delivered to a thread is similar to `pthread_exit`. Outstanding cleanup routines are executed in the context of the target thread, and a status of `-1` is made available to any threads joining with the target thread.

This routine is preferred in implementing an Ada abort statement and any other language- or software-defined construct for requesting thread cancellation.

The results of this routine are unpredictable if the value specified in *thread* refers to a thread that does not currently exist.



## Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The specified thread is invalid.
-1	[ESRCH]	<i>thread</i> does not specify an existing thread.



---

## pthread\_cleanup\_pop

Removes the cleanup handler at the top of the cleanup stack and optionally executes it.

### Syntax

```
pthread_cleanup_pop(  
    execute );
```

Argument	Data Type	Access
<i>execute</i>	Boolean	read

### C Binding

```
int  
pthread_cleanup_pop(  
    int execute);
```

### Arguments

#### **execute**

Integer that specifies whether the cleanup routine in `pthread_cleanup_push` is executed when the thread terminates normally (for example, when `pthread_exit` is called). If the value of *execute* is 0, the routine is executed only if the thread terminates abnormally (for example, if the thread is canceled). If the value is 1 or more, the routine is executed regardless of whether the thread terminates normally or abnormally.

### Description

This routine removes the routine specified in `pthread_cleanup_push` at the top of the calling thread's cleanup stack and executes it if the value specified in *execute* is nonzero.

This routine and `pthread_cleanup_push` are implemented as macros and must appear as statements and in pairs within the same lexical scope. You can think of the `pthread_cleanup_push` macro as expanding to a string whose first character is a left brace (`{`) and `pthread_cleanup_pop` expanding to a string containing the corresponding right brace (`}`).

### Return Values

If an error is detected, it may be indicated by sending the thread a synchronously generated signal.



## pthread\_cleanup\_push

Establishes a cleanup handler to be executed when the thread exits or is canceled.

### Syntax

```
pthread_cleanup_push(  
    *routine,  
    arg );
```

Argument	Data Type	Access
routine	opaque pthread_cleanup_t	read
arg	opaque pthread_addr_t	read

### C Binding

```
int  
pthread_cleanup_push(  
    pthread_cleanup_t *routine,  
    pthread_addr_t arg);
```

### Arguments

**routine**  
Routine executed as the cleanup handler.

**arg**  
Argument executed with the cleanup routine.

### Description

This routine pushes the specified routine onto the calling thread's cleanup stack. The cleanup routine is popped from the stack and executed with the *arg* argument when any of the following actions occur:

- The thread calls `pthread_exit`.
- The thread is canceled.
- The thread calls `pthread_cleanup_pop` and specifies a nonzero value for the *execute* argument.

This routine and `pthread_cleanup_pop` are implemented as macros and must appear as statements and in pairs within the same lexical scope. You can think of the `pthread_cleanup_push` macro as expanding to a string whose first character is a left brace (`{`) and `pthread_cleanup_pop` as expanding to a string containing the corresponding right brace (`}`).

### Return Values

If an error is detected, it may be indicated by sending the thread a synchronously generated signal.



## pthread\_condattr\_create

Creates a condition variable attributes object that can be used to specify the attributes of condition variables when they are created.

### Syntax

```
pthread_condattr_create(  
    attr);
```

Argument	Data Type	Access
attr	opaque pthread_condattr_t	write

### C Binding

```
int  
pthread_condattr_create (  
    pthread_condattr_t *attr);
```

### Arguments

**attr**  
Condition variable attributes object that is created.

### Description

This routine creates a condition variable attributes object that is used to specify the attributes of condition variables when they are created. The condition variable attributes object is initialized with the default value for all of the attributes defined by a given implementation.

When a condition variable attributes object is used to create a condition variable, the values of the individual attributes determine the characteristics of the new object. Attributes objects act like additional arguments to object creation. Changing individual attributes does not affect objects that were previously created using the attributes object.

### Return Values

The created condition variable attributes object is returned to the *attr* argument.

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[ENOMEM]	Insufficient memory exists to create the condition variable attributes object.



## pthread\_condattr\_delete

Deletes a condition variable attributes object.

### Syntax

```
pthread_condattr_delete(
    attr);
```

Argument	Data Type	Access
attr	opaque pthread_condattr_t	read

### C Binding

```
int
pthread_condattr_delete (
pthread_condattr_t *attr);
```

### Arguments

**attr**  
Condition variable attributes object deleted.

### Description

This routine deletes a condition variable attributes object. Call this routine when a condition variable attributes object created by pthread\_condattr\_create will no longer be referenced.

This routine gives permission to reclaim storage for the condition variable attributes object. Condition variables that are created using this attributes object are not affected by the deletion of the condition variable attributes object.

The results of calling this routine are unpredictable if the handle specified by the *attr* argument refers to an attributes object that does not exist.

### Return Values

Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The attributes object specified by <i>attr</i> is invalid.



## pthread\_cond\_broadcast

Wakes all threads that are waiting on a condition variable.

### Syntax

```
pthread_cond_broadcast(  
    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	read

### C Binding

```
int  
pthread_cond_broadcast (  
    pthread_cond_t *cond);
```

### Arguments

**cond**  
Condition variable broadcast.

### Description

This routine wakes all threads waiting on a condition variable. Calling this routine implies that data guarded by the associated mutex has changed so that it might be possible for one or more waiting threads to proceed. If any waiting thread might be able to proceed, call pthread\_cond\_signal.

You can call this routine whether the associated mutex is locked or not.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>cond</i> is invalid.



## pthread\_cond\_destroy

Deletes a condition variable.

### Syntax

```
pthread_cond_destroy(  
    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	read

### C Binding

```
int  
pthread_cond_destroy (  
    pthread_cond_t *cond);
```

### Arguments

**cond**  
Condition variable deleted.

### Description

This routine deletes a condition variable. Call this routine when a condition variable will no longer be referenced. The effect of calling this routine is to give permission to reclaim storage for the condition variable.

The results of this routine are unpredictable if the condition variable specified in *cond* does not exist.

The results of this routine are also unpredictable if there are threads waiting for the specified condition variable to be signaled or broadcasted when it is deleted.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>cond</i> is invalid.
-1	[EBUSY]	A thread is currently executing a <code>pthread_cond_wait</code> or <code>pthread_cond_timedwait</code> on the condition variable specified in <i>cond</i> .



## pthread\_cond\_init

Creates a condition variable.

### Syntax

```
pthread_cond_init(  
    cond,  
    attr );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	write
attr	opaque pthread_condattr_t	read

### C Binding

```
int  
pthread_cond_init (  
    pthread_cond_t *cond,  
    pthread_condattr_t attr);
```

### Arguments

**cond**

Condition variable that is created.

**attr**

Condition variable attributes object that defines the characteristics of the condition variable created. If you specify `pthread_condattr_default`, default attributes are used.

### Description

This routine creates and initializes a condition variable. A condition variable is a synchronization object used in conjunction with a mutex. A mutex controls access to shared data; a condition variable allows threads to wait for that data to enter a defined state.

Condition variables are not owned by a particular thread. Any associated storage is not automatically deallocated when the creating thread terminates.



## Return Values

If an error condition occurs, this routine returns `-1`, the condition variable is not initialized, and the contents of *cond* are undefined. This routine sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EAGAIN]	The system lacks the necessary resources to initialize another condition variable. The system-imposed limit on the total number of condition variables under execution by a single user is exceeded.
-1	[ENOMEM]	Insufficient memory exists to initialize the condition variable.



## pthread\_cond\_signal

Wakes one thread that is waiting on a condition variable.

### Syntax

```
pthread_cond_signal(  
    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	read

### C Binding

```
int  
pthread_cond_signal (  
    pthread_cond_t *cond);
```

### Arguments

**cond**  
Condition variable signaled.

### Description

This routine wakes one thread waiting on a condition variable. Calling this routine implies that data guarded by the associated mutex has changed so that it might be possible for a single waiting thread to proceed. Call this routine when any thread waiting on the specified condition variable might find its predicate true, but only one thread should proceed.

The scheduling policy determines which thread is awakened. For policies SCHED\_FIFO and SCHED\_RR, a blocked thread is chosen in priority order, using first-in/first-out (FIFO) within priorities.

You can call this routine regardless if the associated mutex is locked.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>cond</i> is invalid.



## pthread\_cond\_signal\_int\_np

Wakes one thread that is waiting on a condition variable. This routine can only be called from interrupt level.

### Syntax

```
pthread_cond_signal_int_np(
    cond );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	read

### C Binding

```
int
pthread_cond_signal_int_np(
    pthread_cond_t *cond);
```

### Arguments

**cond**  
Condition variable signaled.

### Description

This routine wakes one thread waiting on a condition variable. It can only be called from a software interrupt handler routine. Calling this routine implies that it might be possible for a single waiting thread to proceed. Call this routine when any thread waiting on the specified condition variable might find its predicate true.

The scheduling policies of the waiting threads determine which thread is awakened. For policies `SCHED_FIFO` and `SCHED_RR`, a blocked thread is chosen in priority order, using first-in/first-out (FIFO) within priorities.

This routine does not cause a thread blocked on a condition variable to resume execution immediately. A thread resumes execution at some time after the interrupt handler returns.

You can call this routine regardless of whether the associated mutex is locked. Never try to lock a mutex from an interrupt handler.

#### Note

This routine allows you to signal a thread from a software interrupt handler. Do not call this routine from noninterrupt code. If you want to signal a thread from the normal noninterrupt level, use `pthread_cond_signal`.



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_cond\_signal\_int\_np

#### Return Values

If an error condition occurs, this routine returns `-1` and sets `errno` to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>cond</i> is invalid.



## pthread\_cond\_sig\_preempt\_int\_np

Wakes one thread that is waiting on a condition variable. This routine can only be called from interrupt level.

### Syntax

```
pthread_cond_sig_preempt_int_np (condition)
pthread_cond_sig_preempt_int_np (condition, scp)
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	read
scp	opaque pthread_addr_t	read

### C Binding

```
void
pthread_cond_sig_preempt_int_np (
pthread_cond_t *condition); void
pthread_cond_sig_preempt_int_np (
pthread_cond_t *condition pthread_addr_t scp);
```

### Arguments

#### cond

Condition variable signaled.

#### scp

UNIX signal control block pointer (this is passed as a parameter to a UNIX signal handler routine).

### Description

This routine wakes one thread waiting on a condition variable. It can only be called from a software interrupt handler routine. Calling this routine implies that it might be possible for a single waiting thread to proceed. Call this routine when any thread waiting on the specified condition variable might find its predicate true.

The scheduling policies of the waiting threads determine which thread is awakened. For policies SCHED\_FIFO and SCHED\_RR, a blocked thread is chosen in priority order, using first-in/first-out (FIFO) within priorities.

You can call this routine when the associated mutex is either locked or unlocked. (Never try to lock a mutex from an interrupt handler.)

#### Note

If the waiting thread has a preemptive scheduling policy and a higher priority than the thread that was running when the interrupt occurred, then the waiting thread will preempt the interrupt routine and begin to run immediately. This is unlike pthread\_cond\_signal\_int\_np which causes the condition variable to be signalled at a safe point after the interrupt has completed.



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_cond\_sig\_preempt\_int\_np

pthread\_cond\_sig\_preempt\_int\_np avoids the possible latency that pthread\_cond\_signal\_int\_np may introduce; however, a side effect of this is that during the call to pthread\_cond\_sig\_preempt\_int\_np other threads may run if a preemption occurs; thus, once an interrupt routine calls pthread\_cond\_sig\_preempt\_int\_np it can no longer rely on any assumptions of exclusivity or atomicity which are typically provided by interrupt routines. Furthermore, once the call to pthread\_cond\_sig\_preempt\_int\_np is made, in addition to other threads running, subsequent interrupts may be delivered at any time as well (that is, they will not be blocked until the current interrupt completes). For this reason, it is recommended that pthread\_cond\_sig\_preempt\_int\_np be called as the last statement in the interrupt routine.

---

#### Note

---

This routine allows you to signal a thread from a software interrupt handler. Do not call this routine from noninterrupt code. If you want to signal a thread from the normal noninterrupt level, use pthread\_cond\_signal.

---

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>cond</i> is invalid.



## pthread\_cond\_timedwait

Causes a thread to wait for a condition variable to be signaled or broadcasted for a specified period of time.

### Syntax

```
pthread_cond_timedwait(
    cond,
    mutex,
    abstime );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	read
mutex	opaque pthread_mutex_t	read
abstime	structure timespec	read

### C Binding

```
int
pthread_cond_timedwait (
    pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    struct timespec *abstime);
```

### Arguments

#### cond

Condition variable waited on.

#### mutex

Mutex associated with the condition variable specified in *cond*.

#### abstime

Absolute time at which the wait expires, if the condition has not been signaled or broadcasted. (See the pthread\_get\_expiration\_np routine, which is used to obtain a value for this argument.)

### Description

This routine causes a thread to wait until one of the following occurs:

- The specified condition variable is signaled or broadcasted.
- The current system clock time is greater than or equal to the time specified by the *abstime* argument.

This routine is identical to pthread\_cond\_wait except that this routine can return before a condition variable is signaled or broadcasted; specifically, when a specified time expires.

If the current time equals or exceeds the expiration time, this routine returns immediately, without causing the current thread to wait. Your code should check the return status whenever this routine returns and take the appropriate action. Otherwise, waiting on the condition variable can become a nonblocking loop.



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_cond\_timedwait

Call this routine after you lock the mutex specified in *mutex*. The results of this routine are unpredictable if this routine is called without first locking the mutex.

#### Return Values

If an error condition occurs, this routine returns `-1` and *errno* is set to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>cond</i> , <i>mutex</i> , or <i>abstime</i> is invalid.  Different mutexes are supplied for concurrent pthread_cond_timedwait operations or pthread_cond_wait operations on the same condition variable.
-1	[EAGAIN]	The time specified by <i>abstime</i> expired.
-1	[EDEADLK]	A deadlock condition is detected.



## pthread\_cond\_wait

Causes a thread to wait for a condition variable to be signaled or broadcasted.

### Syntax

```
pthread_cond_wait(
    cond,
    mutex );
```

Argument	Data Type	Access
cond	opaque pthread_cond_t	read
mutex	opaque pthread_mutex_t	read

### C Binding

```
int
pthread_cond_wait (
    pthread_cond_t *cond,
    pthread_mutex_t *mutex);
```

### Arguments

**cond**  
Condition variable waited on.

**mutex**  
Mutex associated with the condition variable specified in *cond*.

### Description

This routine causes a thread to wait for a condition variable to be signaled or broadcasted. Each condition corresponds to one or more Boolean relations (predicates) based on shared data. The calling thread waits for the data to reach a particular state (for the predicate to become true).

Call this routine after you have locked the mutex specified in *mutex*. The results of this routine are unpredictable if this routine is called without first locking the mutex.

This routine atomically releases the mutex and causes the calling thread to wait on the condition. If the wait is satisfied as a result of some thread calling `pthread_cond_signal` or `pthread_cond_broadcast`, the mutex is reacquired and the routine returns.

A thread that changes the state of storage protected by the mutex in such a way that a predicate associated with a condition variable might now be true must call either `pthread_cond_signal` or `pthread_cond_broadcast` for that condition variable. If neither call is made, any thread waiting on the condition variable continues to wait.

This routine might (with low probability) return when the condition variable has not been signaled or broadcasted. When this occurs, the mutex is reacquired before the routine returns. (To handle this type of situation, enclose this routine in a loop that checks the predicate.)



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_cond\_wait

#### Return Values

If an error condition occurs, this routine returns -1 and *errno* is set to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>cond</i> or <i>mutex</i> is invalid. Different mutexes are supplied for concurrent <i>pthread_cond_wait</i> or <i>pthread_cond_timedwait</i> operations.



## pthread\_create

Creates a thread object and thread.

### Syntax

```
pthread_create(
    thread,
    attr,
    start_routine,
    arg);
```

Argument	Data Type	Access
thread	opaque pthread_t	write
attr	opaque pthread_attr_t	read
start_routine	procedure	read
arg	opaque pthread_addr_t	read

### C Binding

```
int
pthread_create (
    pthread_t *thread,
    pthread_attr_t attr,
    pthread_startroutine_t start_routine,
    pthread_addr_t arg);
```

### Arguments

#### thread

Thread object created.

#### attr

Thread attributes object that defines the characteristics of the thread being created. If you specify pthread\_attr\_default, default attributes are used.

#### start\_routine

Function executed as the new thread's start routine.

#### arg

Address value copied and passed to the thread's start routine.

### Description

This routine creates a thread object and a thread. A **thread** is a single, sequential flow of control within a program. It is the active execution of a designated routine, including any nested routine invocations. A thread object defines and controls the executing thread.

Calling this routine sets into motion the following actions:

- An internal thread object is created to describe the thread.



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_create

- The associated executable thread is created with attributes specified by the *attr* argument (or with default attributes if *pthread\_attr\_default* is specified).
- The *thread* argument receives the new thread.
- The *start\_routine* function is called.

A thread is created in the ready state and therefore might immediately begin executing the function specified by the *start\_routine* argument. The newly created thread may preempt its creator if the new thread follows the *SCHED\_RR* or *SCHED\_FIFO* scheduling policy or has a priority higher than the creating thread, or both. Otherwise, the new thread begins running at its turn, which might also be before *pthread\_create* returns.

The new thread's scheduling policy and priority are, by default, inherited from the creating thread—the scheduling policy and priority set in the attributes object are ignored. To create a thread using the scheduling policy and priority set in the attributes object, you must first disable the inherit scheduling attribute by calling *pthread\_attr\_setinheritsched*.

The *start\_routine* is passed a copy of the *arg* argument. The value of the *arg* argument is specified by the calling application code.

The thread object exists until the *pthread\_detach* routine is called or the thread terminates, whichever occurs last.

Synchronization between the caller of *pthread\_create* and the newly created thread is done through the use of the *pthread\_join* routine (or any other mutexes or condition variables they agree to use).

### Return Values

If an error condition occurs, no thread is created, the contents of *thread* are undefined, and this routine returns *-1* and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EAGAIN]	The system lacks the necessary resources to create another thread.
		The system-imposed limit on the total number of threads under execution by a single user is exceeded.
-1	[ENOMEM]	Insufficient memory exists to create the thread attributes object. This is not a temporary condition.



## pthread\_delay\_np

Causes a thread to delay execution.

### Syntax

```
pthread_delay_np(  
    interval );
```

Argument	Data Type	Access
interval	struct timespec	read

### C Binding

```
extern int  
pthread_delay_np (  
    struct timespec *interval);
```

### Arguments

#### interval

Number of seconds and nanoseconds to delay execution. The value specified for each must be greater than or equal to zero.

### Description

This routine causes a thread to delay execution for a specific period of time. This period ends at the current time plus the specified interval. The thread will not return before the end of the period is reached.

Specifying an interval of 0 seconds and 0 nanoseconds is allowed and can result in the thread giving up the processor or delivering a pending cancel.

The struct timespec structure contains the following two fields:

- tv\_sec is an integer number of seconds
- tv\_nsec is an integer number of nanoseconds

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>interval</i> is invalid.



## pthread\_detach

Marks a thread object for deletion.

### Syntax

```
pthread_detach(  
    thread );
```

Argument	Data Type	Access
thread	opaque pthread_t	read

### C Binding

```
int  
pthread_detach (  
    pthread_t *thread);
```

### Arguments

**thread**

Thread object marked for deletion.

### Description

This routine indicates that storage for the specified thread is reclaimed when the thread terminates. This includes storage for the *thread* argument's return value. If *thread* has not terminated when this routine is called, this routine does not cause it to terminate.

Call this routine when a thread object is no longer referenced. Additionally, call this routine for every thread that is created to ensure that storage for thread objects does not accumulate.

You cannot join with a thread after the thread has been detached.

The results of this routine are unpredictable if the value of *thread* refers to a thread object that does not exist.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>thread</i> is invalid.



## pthread\_equal

Compares one thread identifier to another thread identifier.

### Syntax

```
equal = pthread_equal (thread1, thread2)
```

Argument	Data Type	Access
equal	Boolean	write
thread1	opaque pthread_t	read
thread2	opaque pthread_t	read

### C Binding

```
int
pthread_equal (
pthread_t thread1,
pthread_t thread2);
```

### Arguments

#### equal

Boolean value that specifies whether *thread1* and *thread2* designate the same object.

#### thread1

The first thread identifier to be compared.

#### thread2

The second thread identifier to be compared.

### Description

This routine compares one thread identifier to another thread identifier. (This routine does not check whether the objects that correspond to the identifiers currently exist.) If the identifiers have values indicating that they designate the same object, 1 (true) is returned. If the values do not designate the same object, 0 (false) is returned.

This routine is implemented as a C macro.

### Return Values

Possible return values are as follows:

Return	Error	Description
0		Values of <i>thread1</i> and <i>thread2</i> do not designate the same object.
1		Values of <i>thread1</i> and <i>thread2</i> designate the same object.



## pthread\_exit

Terminates the calling thread.

### Syntax

```
pthread_exit(  
    status );
```

Argument	Data Type	Access
status	opaque pthread_addr_t	read

### C Binding

```
void  
pthread_exit (  
    pthread_addr_t status);
```

### Arguments

#### status

Address value copied and returned to the caller of pthread\_join.

### Description

This routine terminates the calling thread and makes a status value available to any thread that calls pthread\_join and specifies the terminating thread.

An implicit call to pthread\_exit is issued when a thread returns from the start routine that was used to create it. The function's return value serves as the thread's exit status. The process exits when the last running thread calls pthread\_exit.

### Return Values

None



## pthread\_get\_expiration\_np

Obtains a value representing a desired expiration time.

### Syntax

```
pthread_get_expiration_np(  
                                delta,  
                                abstime );
```

Argument	Data Type	Access
<i>delta</i>	struct timespec	read
<i>abstime</i>	struct timespec	write

### C Binding

```
extern int  
pthread_get_expiration_np (  
    struct timespec *delta,  
    struct timespec *abstime);
```

### Arguments

#### **delta**

Number of seconds and nanoseconds to add to the current system time. The result is the time that a timed wait expires.

#### **abstime**

Value representing the expiration time.

### Description

This routine adds a specified interval to the current absolute system time and returns a new absolute time. This new absolute time is used as the expiration time in a call to pthread\_cond\_timedwait.

The struct timespec structure contains the following two fields:

- tv.sec is an integer number of seconds
- tv.nsec is an integer number of nanoseconds

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>delta</i> is invalid.



## pthread\_getprio

Obtains the current priority of a thread.

### Syntax

```
pthread_getprio(  
    thread );
```

Argument	Data Type	Access
thread	opaque pthread_t	read

### C Binding

```
int  
pthread_getprio (  
    pthread_t thread);
```

### Arguments

**thread**  
Thread whose priority is obtained.

### Description

This routine obtains the current priority of a thread. The current priority is different from the initial priority of the thread if the `pthread_setprio` routine is called.

The exact effect of different priority values depends upon the scheduling policy assigned to the thread.

### Return Values

The current priority value of the thread specified in *thread* is returned. See the description of `pthread_setprio` for valid values.

If an error condition occurs, this routine returns `-1` and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Priority value		Successful completion.
-1	[EINVAL]	The value specified by <i>thread</i> is invalid.



## pthread\_getscheduler

Obtains the current scheduling policy of a thread.

### Syntax

```
pthread_getscheduler(  
    thread );
```

Argument	Data Type	Access
thread	opaque pthread_t	read

### C Binding

```
int  
pthread_getscheduler (  
pthread_t thread);
```

### Arguments

**thread**  
Thread whose scheduling policy is obtained.

### Description

This routine obtains the current scheduling policy of a thread. The current scheduling policy of a thread is different from the initial scheduling policy if the pthread\_setscheduler routine is called.

### Return Values

The current scheduling policy value of the thread specified in *thread* is returned. See the description of pthread\_setscheduler for valid values.

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Scheduling policy value		Successful completion.
-1	[EINVAL]	The value specified by <i>thread</i> is invalid.
-1	[ESRCH]	The value specified by <i>thread</i> does not refer to an existing thread.



## pthread\_getspecific

Obtains the per-thread context associated with the specified key.

### Syntax

```
pthread_getspecific(  
    key,  
    value );
```

Argument	Data Type	Access
key	opaque pthread_key_t	read
value	opaque pthread_addr_t	write

### C Binding

```
int  
pthread_getspecific (  
    pthread_key_t key,  
    pthread_addr_t *value);
```

### Arguments

#### key

Context key value that identifies the context value obtained. This key value must be obtained from pthread\_keycreate.

#### value

Address of the current per-thread context value associated with the specified key.

### Description

This routine obtains the per-thread context associated with the specified key for the current thread. If a context has not been defined for the key (that is, pthread\_setspecific has not been successfully executed), NULL is returned in *value*.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The key value is invalid.



## pthread\_join

Causes the calling thread to wait for the termination of a specified thread.

### Syntax

```
pthread_join(
    thread,
    status );
```

Argument	Data Type	Access
thread	opaque pthread_t	read
status	opaque pthread_addr_t	write

### C Binding

```
int
pthread_join (
    pthread_t thread,
    pthread_addr_t *status);
```

### Arguments

#### thread

Thread whose termination is awaited by the caller of this routine.

#### status

Status value of the terminating thread (in other words, when that thread calls pthread\_exit.)

### Description

This routine causes the calling thread to wait for the termination of a specified thread. A call to this routine returns after the specified thread has terminated.

If the thread exits normally, the status value argument is the address that the specified thread generates as its result. The thread's result is normally returned as the value of the *start\_routine* argument in its call to pthread\_create. If the thread does not exit normally, the value of *status* is -1.

Any number of threads can call this routine. All calling threads are awakened when the specified thread terminates.

If the current thread calls this routine, a deadlock can result (if it is not detected by the implementation).

The results of this routine are unpredictable if the value for *thread* refers to a thread object that no longer exists (that is, one that has been detached).



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_join

#### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>thread</i> is invalid.
-1	[ESRCH]	The value specified by <i>thread</i> does not refer to an existing thread.



## pthread\_keycreate

Generates a unique per-thread context key value.

### Syntax

```
pthread_keycreate(
    key,
    destructor );
```

Argument	Data Type	Access
key	opaque pthread_key_t	write
destructor	procedure pthread_destructor_t	read

### C Binding

```
int
pthread_keycreate (
    pthread_key_t *key,
    pthread_destructor_t destructor);
```

### Arguments

#### key

Value of the new per-thread context key.

#### destructor

Procedure called to destroy a context value associated with the created key when the thread terminates.

### Description

This routine generates a unique per-thread context key value. This key value identifies a per-thread context, which is an address of memory generated by the client containing arbitrary data of any size.

Per-thread context allows client software to associate context information with the current thread. (This mechanism can be thought of as a means for a client to add unique fields to the thread control block.)

For example, per-thread context can be used by a language run-time library that needs to associate a language-specific thread-private data structure with an individual thread. The per-thread context routines also provide a portable means of implementing the class of storage called thread-private static, which is needed to support parallel decomposition in the Fortran language.

This routine generates and returns a new key value. The key provides a cell within each thread. Each call to this routine creates a new cell, and each call within a process returns a key value that is unique within an application invocation. Keys must be generated from initialization code that is guaranteed to be called only once within each process. (See the pthread\_once description for more information.)



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_keycreate

When multiple facilities share access to per-thread context, the facilities must agree on the key value that is associated with the context. The key value must be created once and should be stored in a location known to each facility. (Encapsulate key creation and context value setting within a special facility for that purpose.)

An implementation can choose to predefine some number of keys for favored clients, such as certain compilers, run-time libraries, or the debugger.

When a thread terminates, its per-thread context is automatically destroyed; however, the key value remains. For each per-thread context currently associated with the thread, the destructor routine associated with the key value of that context is called. The order in which per-thread context destructors are called at thread termination is undefined.

Upon key creation, the value NULL is associated with the new key in all active threads. Upon thread creation, the value NULL shall be associated with all defined keys in the new thread.

An optional destructor function may be associated with each key value. At thread exit, if a key value has a non-NULL destructor pointer and the thread has a non-NULL value associated with that key, the function pointed to is called with the current associated value as its sole argument. The order of destructor calls is unspecified if more than one destructor exists for a thread when it exits.

If after the destructors have been called for all non-NULL values with associated destructors, there are still some non-NULL values with associated destructores, then the process shall be repeated. If, after at least (PTHREAD\_DESTRUCTOR\_ITERATIONS) iterations of destructor calls for outstanding non-NULL values, there are still some non-values with associated destructors, an application can stop calling destructors, or the application can continue calling destructors until no non-NULL values with the associated destructors exists, even though this might result in an infinite loop.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EAGAIN]	An attempt is made to allocate a key when the key name space is exhausted. This is not a temporary condition.
-1	[ENOMEM]	Insufficient memory exists to create the key.
-1	[EINVAL]	Invalid argument.



---

## pthread\_lock\_global\_np

Locks a global mutex if the global mutex is unlocked. If the global mutex is locked, causes the thread to wait for the global mutex to become available.

### Syntax

```
pthread_lock_global_np( );
```

### C Binding

```
void  
pthread_lock_global_np( );
```

### Arguments

None

### Description

This routine locks the global mutex. If the global mutex is currently held by another thread when a thread calls this routine, the thread waits for the global mutex to become available.

The thread that has locked the global mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the global mutex in the locked state and with the current thread as the global mutex's current owner.

Use the global mutex when calling a library package that is not designed to run in a multithreaded environment. (Unless the documentation for a library function specifically states that it is compatible with multithreading, assume that it is not compatible; in other words, assume it is nonreentrant.)

The global mutex is one lock. Any code that calls any function that isn't known to be reentrant uses the same lock. This prevents dependencies among threads calling library functions and those functions calling other functions, and so on.

The global mutex is a recursive mutex. A thread that has locked the global mutex can relock it without deadlocking. (The locking thread must call `pthread_unlock_global_np` as many times as it called this routine to allow another thread to lock the global mutex.)

Do not call this routine from any software interrupt handler.

### Return Values

None



## pthread\_mutexattr\_create

Creates a mutex attributes object that is used to specify the attributes of mutexes when they are created.

### Syntax

```
pthread_mutexattr_create(  
    attr );
```

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	write

### C Binding

```
int  
pthread_mutexattr_create (  
    pthread_mutexattr_t *attr);
```

### Arguments

**attr**  
Mutex attributes object created.

### Description

This routine creates a mutex attributes object used to specify the attributes of mutexes when they are created. The mutex attributes object is initialized with the default value for all of the attributes defined by a given implementation.

When a mutex attributes object is used to create a mutex, the values of the individual attributes determine the characteristics of the new object. Attributes objects act like additional arguments to object creation. Changing individual attributes does not affect any objects that were previously created using the attributes object.

### Return Values

The created mutex attributes object is returned to the *attr* argument.

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[ENOMEM]	Insufficient memory exists to create the mutex attributes object.



## pthread\_mutexattr\_delete

Deletes a mutex attributes object.

### Syntax

```
pthread_mutexattr_delete(
    attr);
```

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	read

### C Binding

```
int
pthread_mutexattr_delete (
pthread_mutexattr_t *attr);
```

### Arguments

**attr**  
Attributes object deleted.

### Description

This routine deletes a mutex attributes object. Call this routine when a mutex attributes object is no longer referenced by the pthread\_mutexattr\_create routine.

This routine gives permission to reclaim storage for the mutex attributes object. Mutexes that were created using this attributes object are not affected by the deletion of the mutex attributes object.

The results of calling this routine are unpredictable if the attributes object specified in the *attr* argument does not exist.

### Return Values

Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>attr</i> is invalid.



## pthread\_mutexattr\_getkind\_np

Obtains the mutex type attribute used when a mutex is created.

### Syntax

```
pthread_mutexattr_getkind_np(  
    attr );
```

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	read

### C Binding

```
int  
pthread_mutexattr_getkind_np (  
    pthread_mutexattr_t attr);
```

### Arguments

**attr**  
Mutex attributes object whose mutex kind is obtained.

### Description

This routine obtains the mutex type attribute that is used when a mutex is created. See the pthread\_mutexattr\_setkind\_np description for information about mutex type attributes.

### Return Values

On successful completion, this routine returns the mutex kind.

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Mutex type attribute		Successful completion.
-1	[EINVAL]	The value specified by <i>attr</i> is invalid.
-1	[ESRCH]	The value specified by <i>attr</i> does not refer to an existing mutex attributes object.



## pthread\_mutexattr\_setkind\_np

Specifies the mutex type attribute that is used when a mutex is created.

### Syntax

```
pthread_mutexattr_setkind_np(
    attr,
    kind );
```

Argument	Data Type	Access
attr	opaque pthread_mutexattr_t	read
kind	integer	read

### C Binding

```
int
pthread_mutexattr_setkind_np (
    pthread_mutexattr_t *attr,
    int kind);
```

### Arguments

**attr**  
Mutex attributes object modified.

**kind**  
New value for the mutex type attribute. The *kind* argument specifies the type of mutex that is created. Valid values are MUTEX\_FAST\_NP (default), MUTEX\_RECURSIVE\_NP, and MUTEX\_NONRECURSIVE\_NP.

### Description

This routine sets the mutex type attribute that is used when a mutex is created. See Section 2.2.4.1 for information on the types of mutexes.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>attr</i> or <i>kind</i> is invalid.
-1	[ESRCH]	The value specified by <i>attr</i> does not refer to an existing mutex attributes object.



## pthread\_mutex\_destroy

Deletes a mutex.

### Syntax

```
pthread_mutex_destroy(  
    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

### C Binding

```
int  
pthread_mutex_destroy (  
    pthread_mutex_t *mutex);
```

### Arguments

**mutex**  
Mutex deleted.

### Description

This routine deletes a mutex and should be called when a mutex object is no longer referenced. This routine reclaims storage used by the mutex object.

It is illegal to delete a locked mutex.

The results of this routine are unpredictable if the mutex object specified in the *mutex* argument does not currently exist.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EBUSY]	An attempt is made to destroy a locked mutex.
-1	[EINVAL]	The value specified by <i>mutex</i> is invalid.



## pthread\_mutex\_init

Creates a mutex.

### Syntax

```
pthread_mutex_init(
    mutex,
    attr );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	write
attr	opaque pthread_mutexattr_t	read

### C Binding

```
int
pthread_mutex_init (
    pthread_mutex_t *mutex,
    pthread_mutexattr_t attr);
```

### Arguments

**mutex**

Mutex created.

**attr**

Mutex attributes object that defines the characteristics of the created mutex. If you specify pthread\_mutexattr\_default, default attributes are used.

### Description

This routine creates a mutex. A mutex is a synchronization object that allows multiple threads to serialize their access to shared data.

The mutex is created and initialized to the unlocked state.

The created mutex is not automatically deallocated because it is considered shared among multiple threads if the thread that called this routine terminates.

### Return Values

If an error condition occurs, this routine returns -1, the mutex is not initialized, and the contents of *mutex* are undefined. This routine sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EAGAIN]	The system lacks the necessary resources to initialize another mutex.



# POSIX 1003.4a (Draft 4) pthread Routines pthread\_mutex\_init

Return	Error	Description
		The system-imposed limit on the total number of mutexes under execution by a singled user is exceeded.
-1	[ENOMEM]	Insufficient memory exists to initialize the mutex.



## pthread\_mutex\_lock

Locks an unlocked mutex. If the mutex is locked, causes the thread to wait for the mutex to become available.

### Syntax

```
pthread_mutex_lock(  
    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

### C Binding

```
int  
pthread_mutex_lock (  
    pthread_mutex_t *mutex);
```

### Arguments

**mutex**  
Mutex locked.

### Description

This routine locks a mutex and its behavior varies based on the kind of mutex.

If you specified a recursive mutex, the current owner of a mutex can relock the same mutex without blocking. If you specified a nonrecursive mutex and the current owner tries to lock the mutex a second time, the EDEADLK error is reported. If the mutex is locked when a thread calls this routine, the thread waits for the mutex to become available. If you specified a fast mutex, a deadlock can result if the current owner of a mutex calls this routine in an attempt to lock the mutex a second time.

The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it. This routine returns with the mutex in the locked state and with the current thread as the mutex's current owner. See pthread\_mutexattr\_setkind\_np for information about fast, recursive, and nonrecursive mutexes.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>mutex</i> is invalid.
-1	[EDEADLK]	A deadlock condition is detected.



## pthread\_mutex\_trylock

Locks a mutex. If the mutex is already locked, the calling thread does not wait for the mutex to become available.

### Syntax

```
pthread_mutex_trylock(  
    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

### C Binding

```
int  
pthread_mutex_trylock (  
    pthread_mutex_t *mutex);
```

### Arguments

**mutex**  
Mutex locked.

### Description

This routine locks a mutex and its behavior varies based on the kind of mutex. If the specified mutex is locked when a thread calls this routine, the calling thread does not wait for the mutex to become available.

When a thread calls this routine, an attempt is made to immediately lock the mutex. If the mutex is successfully locked, 1 is returned and the current thread is then the mutex's current owner.

If a recursive mutex is owned by the current thread, 1 is returned and the mutex is relocked. (To unlock a recursive mutex, each call to pthread\_mutex\_trylock must be matched by a call to pthread\_mutex\_unlock.) If the mutex is locked by another thread when this routine is called, 0 is returned and the thread does not wait to acquire the lock. If a fast mutex is owned by the current thread, 0 is returned.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
1		Successful completion.
0		The mutex is already locked; therefore, it was not acquired.
-1	[EINVAL]	The value specified by <i>mutex</i> is invalid.



## pthread\_mutex\_unlock

Unlocks a mutex.

### Syntax

```
pthread_mutex_unlock(  
    mutex );
```

Argument	Data Type	Access
mutex	opaque pthread_mutex_t	read

### C Binding

```
int  
pthread_mutex_unlock (  
    pthread_mutex_t *mutex);
```

### Arguments

**mutex**  
Mutex unlocked.

### Description

This routine unlocks a mutex and its behavior varies based on the kind of mutex.

When an owner unlocks a recursive mutex, the lock count is decremented. The mutex remains locked and owned until the count reaches 0. When the lock count reaches 0, or for any other type of mutex, the mutex becomes unlocked with no current owner. If one or more threads are waiting to lock the specified mutex, this routine causes one thread to unblock and try to acquire the mutex. The scheduling policy is used to determine which thread acquires the mutex. For the SCHED\_FIFO and SCHED\_RR policies, a blocked thread is chosen in priority order, using FIFO within priorities.

The results of calling this routine are unpredictable if the mutex specified in *mutex* is unlocked. The results of calling this routine are also unpredictable if the mutex specified in *mutex* is currently owned by a thread other than the calling thread.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The value specified by <i>mutex</i> is invalid.



## pthread\_once

Calls an initialization routine that can be executed by only one thread, a single time.

### Syntax

```
pthread_once(  
    once_block,  
    init_routine );
```

Argument	Data Type	Access
once_block	opaque pthread_once_t	read
init_routine	opaque pthread_initroutine_t	read

### C Binding

```
int  
pthread_once (  
    pthread_once_t *once_block,  
    pthread_initroutine_t init_routine);
```

### Arguments

#### once\_block

Address of a record that defines the one-time initialization code. Each one-time initialization routine must have its own unique pthread\_once\_t.

#### init\_routine

Address of a procedure that performs the initialization. This routine is called only once, regardless of the number of times it and its associated once\_block are passed to pthread\_once.

### Description

This routine calls an initialization routine executed by one thread, a single time. This routine allows you to create your own initialization code that is guaranteed to be run only once, even if called simultaneously by multiple threads or multiple times in the same thread.

For example, a mutex or a per-thread context key must be created exactly once. Calling pthread\_once prevents the code that creates a mutex or per-thread context from being called by multiple threads. Without this routine, the execution must be serialized so that only one thread performs the initialization. Other threads that reach the same point in the code would be delayed until the first thread is finished.

This routine initializes the control record if it has not been initialized and then determines if the one-time initialization routine has executed once. If it has not executed, this routine calls the initialization routine specified in *init\_routine*. If the one-time initialization code has executed once, this routine returns.



---

**Note**

---

If you specify an *init\_routine* that directly or indirectly results in a recursive call to `pthread_once` specifying the same *init\_block* argument, the recursive call will result in a deadlock.

---

The *once\_block* must be declared static (for example, either `extern` or `static` in the C language), and it must be initialized at compile time. In the C language, using `pthread.h` or `pthread_exc.h`, initialize *once\_block* using the `pthread_once_init` macro. In other languages, you must initialize a `pthread_once_t` block to a value of three integer zeroes. In C, that corresponds to the following:

```
static pthread_once_t block = {0,0,0};
```

## Return Values

If an error occurs, this routine returns `-1`. No error values have been specified. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	Invalid argument.



## **pthread\_self**

Obtains the identifier of the current thread.

### **Syntax**

```
pthread_self( );
```

### **C Binding**

```
pthread_t  
pthread_self( );
```

### **Arguments**

None

### **Description**

This routine allows a thread to obtain its own identifier. Use this identifier in calls to `pthread_setprio` and `pthread_setscheduler`.

This value becomes meaningless when the thread object is deleted—that is, when the thread has terminated its execution and `pthread_detach` has been called.

### **Return Values**

Returns the identifier of the calling thread to `pthread_t`.



## pthread\_setasynccancel

Enables or disables the current thread's asynchronous cancelability.

### Syntax

```
old_state = pthread_setasynccancel(
                                state );
```

Argument	Data Type	Access
state	integer	read

### C Binding

```
int
pthread_setasynccancel (
int state);
```

### Arguments

#### state

State of asynchronous cancelability to set for the calling thread. Valid values are as follows:

Value	Description
CANCEL_ON	Asynchronous cancelability is enabled.
CANCEL_OFF	Asynchronous cancelability is disabled.

### Description

This routine enables or disables the current thread's asynchronous cancelability and returns the previous cancelability state.

When general cancelability is set to CANCEL\_OFF, a cancel cannot be delivered to the thread, even if a cancelable routine is called or asynchronous cancelability is enabled. When general cancelability is set to CANCEL\_ON, cancelability depends on the state of the thread's asynchronous cancelability. When general cancelability is set to CANCEL\_ON and asynchronous cancelability is set to CANCEL\_OFF, the thread can only receive a cancel at specific cancellation points (for example, condition waits, thread joins, and calls to pthread\_testcancel.) If both general cancelability and asynchronous cancelability are set to CANCEL\_ON, the thread can be canceled at any point in its execution.

When a thread is created, the default asynchronous cancelability state is CANCEL\_OFF.

If you call this routine to enable asynchronous cancelability, first call it specifying CANCEL\_OFF to reliably obtain the previous cancellation state. Next, call this routine a second time specifying CANCEL\_ON to enable asynchronous cancelability and ignore the result of this second call. The result of the second call is not reliable because an asynchronous cancel may occur before the routine returns. The previous state of asynchronous delivery can be restored later by another call



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_setasynccancel

to this routine specifying the value returned by the call that specified `CANCEL_OFF`. See Example 2-1.

If asynchronous cancelability is enabled, do not call any routine unless it is explicitly documented as safe to be called with asynchronous cancelability enabled. Note that none of the general run-time libraries and none of the DECthreads libraries are safe except for `pthread_setasynccancel` and `cma_alert_restore`.

### Return Values

On successful completion, this routine returns the previous state of asynchronous cancelability.

If an error condition occurs, this routine returns `-1` and sets `errno` to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The specified state is not <code>CANCEL_ON</code> or <code>CANCEL_OFF</code> .



## pthread\_setcancel

Enables or disables the current thread's general cancelability.

### Syntax

```
int pthread_setcancel(
    state );
```

Argument	Data Type	Access
state	integer	read

### C Binding

```
int
pthread_setcancel (
    int state);
```

### Arguments

#### state

State of general cancelability to set for the calling thread. Valid values are as follows:

Value	Description
CANCEL_ON	Asynchronous cancelability is enabled.
CANCEL_OFF	Asynchronous cancelability is disabled.

### Description

This routine enables or disables the current thread's general cancelability and returns the previous cancelability state.

When general cancelability is set to CANCEL\_OFF, a cancel cannot be delivered to the thread, even if a cancelable routine is called or asynchronous cancelability is enabled.

When a thread is created, the default general cancelability state is CANCEL\_ON.

#### Possible Dangers of Disabling Cancelability

The most important use of cancels is to ensure that indefinite wait operations are terminated. For example, a thread waiting on some network connection, which may take days to respond (or may never respond), should be made cancelable.

However, when cancelability is disabled, no routine is cancelable. As a result, the user is unable to cancel the operation.

When disabling cancelability, be sure that no long waits can occur or that it is necessary for other reasons to defer cancels around that particular region of code.



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_setcancel

#### Return Values

On successful completion, this routine returns the previous state of general cancelability.

If an error condition occurs, this routine returns `-1` and sets `errno` to the corresponding error value. Possible return values are as follows:

Return	Error	Description
CANCEL_ON		Successful completion.
CANCEL_OFF		Successful completion.
-1	[EINVAL]	The specified state is not CANCEL_ON or CANCEL_OFF.



## pthread\_setprio

Changes the current priority of a thread.

### Syntax

```
pthread_setprio(  
    thread,  
    priority );
```

Argument	Data Type	Access
thread	opaque pthread_t	read
priority	integer	read

### C Binding

```
int  
pthread_setprio (  
    pthread_t thread,  
    int priority);
```

### Arguments

#### thread

Thread whose priority is changed.

#### priority

New priority value of the thread specified in *thread*. The priority value is dependent upon scheduling policy. Valid values fall within one of the following ranges.

Low	High
PRI_FIFO_MIN	PRI_FIFO_MAX
PRI_RR_MIN	PRI_RR_MAX
PRI_OTHER_MIN	PRI_OTHER_MAX
PRI_FG_MIN_NP	PRI_FG_MAX_NP
PRI_BG_MIN_NP	PRI_BG_MAX_NP

If you create a new thread without specifying a threads attributes object that contains a changed priority attribute, the default priority of the newly created thread is the midpoint between PRI\_OTHER\_MIN and PRI\_OTHER\_MAX (the midpoint between the minimum and the maximum for the SCHED\_OTHER policy). (Section 2.7 describes how to specify priorities between the minimum and maximum values.)



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_setprio

#### Description

This routine changes the current priority of a thread. A thread can change its own priority using the identifier returned by `pthread_self`.

Changing the priority of a thread can cause it to start executing or be preempted by another thread. The effect of setting different priority values depends on the scheduling priority assigned to the thread. The initial scheduling priority is set by calling the `pthread_attr_setprio` routine.

An application should specify priority only to express the urgency of executing the thread relative to other threads. Priority should not be used to control mutual exclusion when accessing shared data. With a sufficient number of processors executing, all ready threads, regardless of priority, can be executing simultaneously.

The `pthread_attr_setprio` routine sets the priority attribute that is used to establish the priority of a new thread when it is created. However, `pthread_setprio` changes the priority of an existing thread.

#### Return Values

The previous priority of the thread specified in *thread* is returned.

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Priority value		Successful completion.
-1	[EINVAL]	The value specified by <i>priority</i> is invalid.
-1	[ENOTSUP]	An attempt is made to set the policy to an unsupported value.
-1	[ESRCH]	The value specified by <i>thread</i> does not refer to an existing thread.
-1	[EPERM]	The caller does not have the appropriate privileges to set the priority of the specified thread.



## pthread\_setscheduler

Changes the current scheduling policy and priority of a thread.

### Syntax

```
pthread_setscheduler(  
    thread,  
    scheduler,  
    priority );
```

Argument	Data Type	Access
thread	opaque pthread_t	read
scheduler	integer	read
priority	integer	read

### C Binding

```
int  
pthread_setscheduler (  
    pthread_t thread,  
    int scheduler,  
    int priority);
```

### Arguments

#### thread

Thread whose scheduling policy is to be changed.

#### scheduler

New scheduling policy value for the thread specified in *thread*. (Policies listed on the same line are equivalent.) Valid values are as follows:

```
SCHED_FIFO  
SCHED_RR  
SCHED_FG_NP or SCHED_OTHER  
SCHED_BG_NP
```

See Section 2.2.3.2 for a description of the scheduling policies.

#### priority

New priority value of the thread specified in *thread*. The priority attribute is dependent upon scheduling policy. Valid values fall within one of the following ranges.

Low	High
PRI_FIFO_MIN	PRI_FIFO_MAX
PRI_RR_MIN	PRI_RR_MAX
PRI_OTHER_MIN	PRI_OTHER_MAX



## POSIX 1003.4a (Draft 4) pthread Routines

### pthread\_setscheduler

Low	High
PRI_FG_MIN_NP	PRI_FG_MAX_NP
PRI_BG_MIN_NP	PRI_BG_MAX_NP

If you create a new thread without specifying a thread's attributes object that contains a changed priority attribute, the default priority of the newly created thread is the midpoint between `PRI_OTHER_MIN` and `PRI_OTHER_MAX` (the midpoint between the minimum and the maximum for the `SCHED_OTHER` policy). (Section 2.7 describes how to specify priorities between the minimum and maximum values.)

### Description

This routine changes the current scheduling policy and priority of a thread. Call this routine to change both the priority and scheduling policy of a thread at the same time. To change only the priority, call the `pthread_setprio` routine.

A thread changes its own scheduling policy and priority by using the identifier returned by `pthread_self`. Changing the scheduling policy or priority, or both, of a thread can cause it to start executing or to be preempted by another thread.

This routine is different from `pthread_attr_setprio` and `pthread_attr_setsched` because those routines set the priority and scheduling policy attributes that are used to establish the priority and scheduling policy of a new thread when it is created. This routine, however, changes the priority and scheduling policy of an existing thread.

### Return Values

The previous policy of the thread specified in *thread* is returned.

If an error condition occurs, this routine returns `-1` and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
Scheduling policy value		Successful completion.
-1	[EINVAL]	The value specified by <i>scheduler</i> is invalid.
-1	[ENOTSUP]	An attempt is made to set the policy to an unsupported value.
-1	[ESRCH]	The value specified by <i>thread</i> does not refer to an existing thread.
-1	[EPERM]	The caller does not have the appropriate privileges to set the priority of the specified thread.



## pthread\_setspecific

Sets the per-thread context associated with the specified key for the current thread.

### Syntax

```
pthread_setspecific(
    key,
    value );
```

Argument	Data Type	Access
key	opaque pthread_key_t	read
value	opaque pthread_addr_t	read

### C Binding

```
int
pthread_setspecific (
    pthread_key_t key,
    pthread_addr_t value);
```

### Arguments

#### key

Context key value that uniquely identifies the context cell to receive *value*. This key value must be obtained from pthread\_keycreate.

#### value

Address containing data associated with the specified key for the current thread; this is the per-thread context.

### Description

This routine sets the per-thread context associated with the specified key for the current thread. If a context is defined for the key in this thread (the current value is not null), the new value is substituted for it.

Different threads can bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that are reserved for use by the calling thread.

### Return Values

If an error condition occurs, this routine returns -1 and sets *errno* to the corresponding error value. Possible return values are as follows:

Return	Error	Description
0		Successful completion.
-1	[EINVAL]	The key value is invalid.



## pthread\_testcancel

Requests delivery of a pending cancel to the current thread.

### Syntax

```
pthread_testcancel( );
```

### C Binding

```
void  
pthread_testcancel ( );
```

### Arguments

None

### Description

This routine requests delivery of a pending cancel to the current thread. The cancel is delivered only if a cancel is pending for the current thread and general cancel delivery is not currently disabled. (A thread disables delivery of cancels to itself by calling pthread\_setcancel.)

This routine, when called within very long loops, ensures that a pending cancel is noticed within a reasonable amount of time.

### Return Values

None



---

## pthread\_unlock\_global\_np

Unlocks a global mutex.

### Syntax

```
pthread_unlock_global_np( );
```

### C Binding

```
void  
pthread_unlock_global_np ( );
```

### Arguments

None

### Description

This routine unlocks the global mutex when each call to `pthread_lock_global_np` has been matched by a call to this routine. For example, if you called `pthread_lock_global_np` three times, `pthread_unlock_global_np` unlocks the global mutex when you call it the third time. If no threads are waiting for the global mutex, it becomes unlocked with no current owner. If one or more threads are waiting to lock the global mutex, this routine causes one thread to unblock and try to acquire the mutex. The scheduling policy is used to determine which thread acquires the global mutex. For the policies `SCHED_FIFO` and `SCHED_RR`, a blocked thread is chosen in priority order, using FIFO within priorities.

The results of calling this routine are unpredictable if the global mutex is already unlocked. The results of calling this routine are also unpredictable if the global mutex is owned by a thread other than the calling thread.

Do not call this routine from any software interrupt handler.

### Return Values

None



---

## pthread\_yield

Notifies the scheduler that the current thread is willing to release its processor to other threads of the same or higher priority.

### Syntax

```
pthread_yield( );
```

### C Binding

```
void  
pthread_yield ( );
```

### Arguments

None

### Description

This routine notifies the thread scheduler that the current thread is willing to release its processor to other threads of equivalent or greater scheduling precedence. (A thread generally will release its processor to a thread of a greater scheduling precedence without calling this routine.) If no other threads of equivalent or greater scheduling precedence are ready to execute, the thread continues.

This routine can allow knowledge of the details of an application to be used to improve its performance. If a thread does not call `pthread_yield`, other threads may be given the opportunity to run at arbitrary points (possibly even when the interrupted thread holds a required resource). By making strategic calls to `pthread_yield`, other threads can be given the opportunity to run when the resources are free. This improves performance by reducing contention for the resource.

As a general guideline, consider calling this routine after a thread has released a resource (such as a mutex) which is heavily contended for by other threads. This can be especially important if the program is running on a uniprocessor machine, or if the thread acquires and releases the resource inside a tight loop.

Use this routine carefully and sparingly, because misuse can cause unnecessary context switching which will increase overhead and actually degrade performance. For example, it is counter-productive for a thread to yield while it holds a resource that the threads to which it is yielding will need. Likewise, it is pointless to yield unless there is likely to be another thread that is ready to run.

### Return Values

None



---

## Glossary

### **alert**

*See cancel.*

### **alertable routine**

*See cancelable routine.*

### **AST**

Mechanism that signals an asynchronous event to a process.

### **asynchronous cancelability**

If enabled, allows a thread to receive a cancellation request at any time (not only at cancellation points). *See also* general cancelability.

### **asynchronous signal**

Signal that is the result of an event that is external to the process and is delivered at any point in a thread's execution when such an event occurs. *See also* synchronous signal.

### **atomic queue**

DECthreads Library object that can be used to communicate information among threads or among routines in a single thread.

### **attributes**

Individual components of the attributes object. Attributes specify detailed properties about the objects to be created. *See also* attributes object.

### **attributes object**

Object used to describe DECthreads objects (thread, mutex, condition variable, queue, or attributes object). This description consists of the individual attribute values that are used to create an object. *See also* attributes.

### **cancel**

Mechanism by which one thread requests termination of another thread (or itself).

### **cancelable routine**

Routine in which a cancel may be delivered.

### **cancellation point**

DECthreads routine that, when called by a routine, can determine whether a cancel is pending for that routine, and if so, can deliver the cancel.



**condition variable**

Object that allows a thread to block its own execution until some shared data reaches a particular state.

**deadlock**

Condition involving one or more threads and a set of one or more resources in which each of the threads is blocked waiting for one of the resources and all of the resources are held by the threads such that none of the threads can continue. For example, a thread will enter a self-deadlock when it attempts to lock a "fast" mutex a second time. Likewise, two threads will enter a deadlock when each attempts to lock a second mutex that is already held by the other. The introduction of additional threads and synchronization objects allows for more complex deadlock configurations.

**dynamic memory**

Memory that is allocated by the program as a result of a call to some memory management function, and that is referenced through pointer variables. *See also* static memory and stack memory.

**exception**

Object that describes an error condition.

**exception scope**

Block of code where exceptions are handled.

**fast mutex**

A kind of mutex that can be locked exactly once by a thread. It does not perform error checks. If a thread tries to lock the mutex again without first unlocking it, the thread waits for itself to release the lock and deadlocks. "Fast" refers to a streamlined implementation that provides the best performance. *See also* mutex.

**general cancelability**

If enabled, allows a thread to receive a cancellation request at specific cancellation points. If disabled, the thread cannot be canceled. *See also* asynchronous cancelability.

**global lock**

Single recursive mutex provided by DECthreads for use by all threads in a process when calling routines or code that is not thread-safe to ensure serialized, exclusive access to the unsafe code.

**guard area**

Area at the end of the thread stack that is inaccessible to the thread. This helps prevent or detect overflow of the thread's stack.

**guardsize attribute**

Minimum size (in bytes) of the guard area for the stack of a thread.

**handle**

Storage, similar to a pointer, that refers to a specific DECthreads object.



**inherit scheduling attribute**

Attribute that specifies whether a newly created thread inherits the scheduling attributes (scheduling priority and policy) of the creating thread or uses the scheduling attributes stored in the attributes object. *See also* thread attributes object.

**lifetime**

Length of time memory is allocated for a particular purpose.

**multithreaded programming**

Division of a program into multiple threads that execute concurrently.

**mutex**

Meaning mutual exclusion, an object that multiple threads use to ensure the integrity of a shared resource that they access (most commonly shared data) by allowing only one thread to access it at a time. *See also* fast mutex, nonrecursive mutex, and recursive mutex.

**mutex attributes object**

Attribute that allows you to specify values for mutex attributes when you create a mutex.

**mutex kind attribute**

Attribute that specifies whether a mutex is *fast*, *recursive*, or *nonrecursive*.

**nonrecursive mutex**

Mutex that can be locked exactly once by a thread, like a fast mutex. If a thread tries to lock the mutex again without first unlocking it, the thread receives an error instead of deadlocking. *See also* mutex, fast mutex, and deadlock.

**nonterminating signal**

Signal that does not result in the termination of the process by default. *See also* terminating signal.

**per-thread context**

*See* thread-specific data.

**predicate**

Boolean expression that defines a particular state of shared data; threads wait on a condition variable for shared data to enter the defined state. *See also* condition variable.

**priority inversion**

Occurs when interaction among three or more threads blocks the highest-priority thread from executing until after the lowest-priority thread can execute.

**queuesize attribute**

Number of elements allowed on a queue. *See also* atomic queue.

**race condition**

Occurs when two or more threads perform an operation, and the result of the operation depends on unpredictable timing factors.



**recursive mutex**

Mutex that can be locked more than once by a given thread without causing a deadlock. The thread must call the `cma_mutex_unlock` or `pthread_mutex_unlock` routine the same number of times that it called the `cma_mutex_lock` or `pthread_mutex_lock` routine before another thread can lock the mutex. *See also* mutex and deadlock.

**scheduling policy attribute**

Attribute that describes how the thread is scheduled for execution relative to the other threads in the program. *See also* thread attributes object.

**scheduling precedence**

The set of characteristics of threads and the DECthreads scheduling algorithm that, in combination, determine which thread will be allowed to run when a scheduling decision is made. Scheduling decisions are made when a thread becomes ready to run (for example, when a mutex on which it was waiting is unlocked or a condition variable on which it was waiting is signaled or broadcasted), or when a thread is blocked (for example, when it attempts to lock a locked mutex, or when it waits on a condition variable).

**scheduling priority attribute**

Attribute that specifies the execution priority of a thread, expressed relative to other threads in the same policy. *See also* thread attributes object.

**scope**

Areas of a program where code can access memory.

**software interrupt handler**

A routine that is executed in response to an interrupt generated by the operating system or equivalent support software. For example: an AST service routine handles interrupts on OpenVMS systems; a signal handler routine handles interrupts on UNIX systems.

**stacksize attribute**

Minimum size (in bytes) of the memory required for a thread's stack.

**stack memory**

Memory that is allocated from a thread's stack area at run time by code generated by the language compiler, generally when a routine is initially called. *See also* dynamic memory and static memory.

**static memory**

Any variable that is permanently allocated at a particular address for the life of the program. *See also* dynamic memory and stack memory.

**synchronous signal**

Signal that is the result of an event that occurs inside a process and is delivered synchronously with respect to that event. *See also* asynchronous signal.

**terminating signal**

Signal that results in the termination of the process by default. *See also* nonterminating signal.



**thread**

Single, sequential flow of control within a program. Within a single thread, there is a single point of execution.

**thread attributes object**

Object that allows you to specify values for thread attributes when you create a thread.

**thread-reentrant**

Routine that functions normally despite being called simultaneously or sequentially in different threads.

**thread-safe**

Routine that can be called simultaneously from multiple threads without risk of corruption.

**thread-specific data**

User-specified fields of arbitrary data that can be added to a thread's context.

**timeslicing**

Mechanism that ensures that every thread is allowed time to execute by preempting running threads at fixed intervals.



1950

George Washington University, Washington, D.C. 20057  
1000 14th Street, N.W.

1000 14th Street, N.W.

1000 14th Street, N.W. Washington, D.C. 20057  
1000 14th Street, N.W.

1000 14th Street, N.W.

1000 14th Street, N.W. Washington, D.C. 20057  
1000 14th Street, N.W.

1000 14th Street, N.W.

1000 14th Street, N.W. Washington, D.C. 20057  
1000 14th Street, N.W.

1000 14th Street, N.W.

1000 14th Street, N.W. Washington, D.C. 20057  
1000 14th Street, N.W.

1000 14th Street, N.W.

1000 14th Street, N.W. Washington, D.C. 20057  
1000 14th Street, N.W.



---

# Index

64-Bit addressing  
on OpenVMS Alpha, B-4

## A

### Access

shared resources, 4-2

### Ada compiler

generating reentrant code, 4-1

### Alert

asynchronous delivery and exception handlers,  
E-13

delivery, E-67

disabling asynchronous delivery of, E-11

disabling delivery of, E-12

enabling asynchronous delivery of, E-13

enabling delivery of, E-15

requesting delivery of, E-17

sending to a thread, E-67

using asynchronous delivery with external  
routines, E-11

### Alert delivery state

restoring, E-16

### Alertable

ensuring for matrix multiplication, E-13

### API errors, reporting, 3-9

### ASTs, G-39

### ASTs (asynchronous system traps)

restrictions on use, B-3

### Asynchronous cancelability, 2-14

### Asynchronous execution

designing code for, 3-1

### Asynchronous programming techniques

using in a multithreaded program, A-5

### Asynchronous signals, A-4

### Asynchronous system traps

See ASTs

### Atomic queue, E-4

### Attributes

See also Attributes object

condition variable, 2-6

detachstate, pthread-6, pthread-20, G-8, G-14

guard size, 2-6, pthread-8, pthread-22, E-21,

E-27, G-9, G-16

inherit scheduling, 2-4

mutex kind, E-23, E-30

mutex type, 2-6

### Attributes (cont'd)

priority, E-24, E-31, G-11, G-19

queue size, F-4, F-5

scheduling, pthread-10, pthread-24, E-22,  
E-28, G-10, G-17

scheduling parameters, pthread-12

scheduling policy, 2-4, pthread-14,

pthread-28, E-25, E-33, G-12, G-21

scheduling policy attribute parameters,  
pthread-26

scheduling priority, 2-5

stack size, 2-6, pthread-16, pthread-30, E-26,  
E-35, G-13, G-23

thread, 2-4

### Attributes object

creating, 2-3, E-18, G-5

creating and initializing, pthread-18

definition of, 2-3

deleting, 2-3, E-20

## B

### Background scheduling, 2-4

### BLISS compiler

generating reentrant code, 4-1

### Boss/worker model, 1-5

work queue variation, 1-5

### Broadcasting a wakeup, pthread-37, tis-3, E-36, G-32

## C

### C compiler

generating reentrant code, 4-1

### Cancel

asynchronous delivery and exception handlers,  
pthread-104, G-73

delivery, pthread-32, G-26

enabling and disabling asynchronous delivery  
of, pthread-104, G-73

enabling and disabling delivery of,  
pthread-102, tis-33, G-75

obtaining noncancelable versions of cancelable  
routines, tis-33, G-75

possible dangers of disabling, pthread-102,  
tis-33, G-75

requesting delivery of, pthread-113, tis-37,  
G-82



## Cancel (cont'd)

- sending to a thread, pthread-32, G-26
- setting current thread state, pthread-102
- setting current thread type, pthread-104

## Cancelability

- asynchronous, 2-14, pthread-104, G-73
- general, tis-33, G-75
- state, 2-14
- type, 2-14

## Cancelability state

- thread, pthread-102

## Cancelability type

- thread, pthread-104

## Canceling a thread, 2-14

- See Thread, canceling

## CATCH exception, 5-4

## CATCH\_ALL exception, 5-8

## Characteristics of created condition variable

- specifying, pthread-52, G-30

## Characteristics of created mutex

- specifying, G-60

## Characteristics of created object

- specifying, pthread-18, E-18, G-5

## Cleanup routine

- establishing, pthread-35, G-29
- executing, pthread-34, G-28

## Client, 1-1

## cma.h, A-2, B-2

## cma\_debug, E-46

## cma\_t\_once data structure, E-62

## COBOL compiler

- generating nonreentrant code, 4-1

## Combination model, 1-6

## Comparing two handles, E-50

## Compilers

- generating nonreentrant code, 4-1
- generating reentrant code, 4-1

## Compiling for OpenVMS images, B-2

## Condition handlers

- declaring, B-4

## Condition variable, 2-9

- comparing to mutex, 3-7
- creating, pthread-41, tis-6, E-37, G-34
- definition of, pthread-41, tis-6, G-34
- deleting, pthread-39, tis-4, E-39, G-33
- signaling, 3-9
- waiting for, pthread-49, tis-9, E-44, G-43
- waiting for a specified time, pthread-47, E-42, G-41

## Condition variable attributes, 2-6

## Condition variable attributes object

- creating, G-30

- deleting, pthread-51, G-31

- initializing, pthread-52

## Condition variables

- thread-safe, 4-2

## Context

- generating key value for, pthread-74, tis-12, E-52, G-57

- obtaining, pthread-71, tis-11, E-54, G-54
- per-thread, 2-13

- setting, pthread-109, tis-35, E-55, G-81

- uses, pthread-74

- uses for, tis-12, E-52, G-57

## Copying a handle, E-49

## Creating

- attributes object, E-18

- condition variable attributes object, pthread-52, G-30

- mutex attributes object, pthread-95, G-60

- thread attributes object, pthread-18, G-5

## Creating a condition variable, pthread-41, tis-6, E-37, G-34

## Creating a mutex, pthread-84, tis-18, E-57, G-65

## Creating a thread, pthread-54, E-69, G-45

- detached creation attribute, pthread-6, pthread-20, G-8, G-14

- guardsize attribute, pthread-8, pthread-22, E-21, E-27, G-9, G-16

- inherit scheduling attribute, pthread-10, pthread-24, E-22, E-28, G-10, G-17

- priority attribute, E-24, E-31, G-11, G-19

- scheduling parameters attribute, pthread-12
- scheduling policy attribute, pthread-14, pthread-28, E-25, E-33, G-12, G-21

- scheduling policy attribute parameters, pthread-26

- stacksize attribute, pthread-16, pthread-30, E-26, E-35, G-13, G-23

## Creating detached threads, pthread-6, pthread-20, G-8, G-14

## Creating per-thread context key value, pthread-74, tis-12, E-52, G-57

## D

### Data

- thread-specific, 2-13, 4-2

### Data structures

- cma\_t\_once, E-62

- pthread\_once\_t, pthread-99, tis-23, G-70

### Deadlocks

- how to avoid, 3-8

### Debugging a multithreaded program, pthread-57, pthread-58, E-46, E-47

### Debugging threads

- on systems based on Windows NT software, D-10

### Declaring a condition handler, B-4

### DECthreads Library routines, F-1



Delaying execution of a thread, pthread-60, E-48, G-47

#### Deleting

attributes object, E-20

condition variable attributes object,  
pthread-51, G-31

mutex attributes object, pthread-92, G-61

thread attributes object, G-7

Deleting a condition variable, pthread-39, tis-4,  
E-39, G-33

Deleting a mutex, tis-16, E-58, G-64

Deleting a thread, pthread-61, E-71, G-48

#### Deleting key

pthread, pthread-76, tis-14

#### Delivery of alerts

disabling, E-12

disabling asynchronous delivery of, E-11

enabling, E-15

enabling asynchronous delivery of, E-13

requesting, E-17

#### Delivery of cancels

enabling and disabling, tis-33, G-75

enabling and disabling asynchronous delivery  
of, G-73

requesting, pthread-113, tis-37, G-82

Dequeue, E-4

#### Destroying

readers/writer locks, tis-30

thread attributes object, pthread-5

Destroying a mutex, pthread-82

Detaching thread attribute, pthread-20

Digital Proprietary Interfaces: cma routines, E-1

Digital Proprietary Interfaces: tis routines, tis-3

Digital UNIX, A-2

Disabling asynchronous delivery of alerts, E-11

Disabling asynchronous delivery of cancels, G-73

#### Distributed system

using threads in, 1-1

DOS File Allocation Table (FAT) file system, C-2

Dynamic memory, 3-5

## E

Enabling asynchronous delivery of alerts, E-13

Enabling asynchronous delivery of cancels, G-73

Enqueue, E-4

Error termination of a thread, pthread-54, E-69,  
E-72, G-45

Errorcheck mutex, 2-8

Errors, reporting, 3-9

#### Example programs

prime number search, 6-1

#### Exceptions

CATCH, 5-4

catching, 5-4

CATCH\_ALL, 5-8

condition handler causing to fail, B-4

declaring and initializing, 5-3

#### Exceptions (cont'd)

defining a region of code to catch, 5-4

defining epilogue actions, 5-6

definition of, 5-1

determining current, 5-6

ENDTRY, 5-4

exporting error status, 5-7

FINALLY, 5-6, 5-10

importing error status, 5-6

introduction to, 5-1

matching, 5-7

naming convention for, 5-10

pthread\_exc\_get\_status\_np, 5-7

pthread\_exc\_matches\_np, 5-7

pthread\_exc\_report\_np, 5-7

pthread\_exc\_set\_status\_np, 5-6

raising, 5-3

reporting, 5-7

RERAISE, 5-5, 5-8, 5-11

reraising, 5-5

rules for modular use of, 5-10

signals reported as, A-6

table listing pthread exceptions and meanings,  
5-11

THIS\_CATCH, 5-6

TRY, 5-4

#### Expiration time

obtaining, pthread-66, E-83, G-51

## F

Fast mutex, 2-7, E-30, G-63

FIFO (first-in/first-out) scheduling, 2-4

FINALLY exception, 5-6, 5-10

#### Fork

handling, pthread-3

## G

Global lock, 3-4

using to avoid nonreentrant software, 3-4

#### Global mutex

locking, pthread-80, tis-15, E-56, G-59

unlocking, pthread-114, tis-38, E-84, G-83

Guardsize attribute, 2-6, pthread-22, E-21,  
E-27, G-16

obtaining, pthread-8, G-9

## H

#### Handlers

declaring a condition handler, B-4

#### Handles, E-1

assigning to an object, E-49

comparing, E-50

copying, E-49

obtaining for thread, E-76



Header files, A-2, B-2

## I

### Identifiers

comparing, pthread-63, G-49

### Images

compiling for OpenVMS, B-2

linking to OpenVMS, B-3

### Inherit scheduling attribute, 2-4, E-22

obtaining, pthread-10, G-10

usefulness, pthread-24, E-28, G-17

### Initialization

one-time, pthread-99, tis-23, E-62, G-70

### Initialization routines

one-time, 2-13

### Initializing

condition variable, pthread-41, tis-6, E-37, G-34

readers/writer lock, tis-31

thread attributes object, pthread-18

threads routines, E-51

### INT suffix on DECthreads routines, B-3

### Internal errors, reporting, 3-9

### Interrupt, E-41, G-37, G-39

### Interrupt handler

inserting a queue element from, F-13

waking a thread, pthread-45

## K

### Kernel threads

effects of context switching, 3-1

### Key value

deleting, pthread-76, tis-14

generating for per-thread context, pthread-74, tis-12, E-52, G-57

obtaining per-thread context for, pthread-71, tis-11, E-54, G-54

setting per-thread context for, pthread-109, tis-35, E-55, G-81

## L

### Lifetime

definition of, 3-4

### Linking to OpenVMS images, B-3

### Lock

global, 3-4

### Locking

global mutex, pthread-80, tis-15, E-56, G-59

readers locks, tis-26, tis-27

writer lock, tis-39, tis-40

Locking a mutex, pthread-86, pthread-88, tis-20, tis-21, E-59, E-60, G-67, G-68

### Locks

readers/writer, 4-2

## M

### Memory

dynamic, 3-5

locking, tis-26, tis-27, tis-30, tis-31, tis-39, tis-40, tis-42

setting for a thread's stack, 2-6

stack, 3-5

static, 3-5

types of, 3-5

unlocking, tis-29

### Multithreaded programming

introduction, 1-1

potential problems, 1-7

complexity, 1-7

deadlocks, 3-8

nonreentrant routines, 1-7

priority inversion, 3-7

race conditions, 3-8

software models, 1-5

boss/worker, 1-5

combination, 1-6

pipelining, 1-6

work crew, 1-5

### Mutex, 2-7

comparing to condition variable, 3-7

creating, tis-18, E-57, G-65

definition of, pthread-84, tis-18, G-65

deleting, tis-16, E-58, G-64

destroying, pthread-82

errorcheck, 2-8

fast, E-30, G-63

global lock, pthread-80

global unlock, pthread-114

initializing, pthread-84

locking, pthread-86, pthread-88, tis-20, tis-21, E-59, E-60, G-67, G-68

locking before signaling condition variable, 3-9

nonrecursive, pthread-97, G-63

normal, 2-7, pthread-97

obtaining kind, E-23

recursive, 2-7, pthread-97, E-30, G-63

setting kind, E-30

types of, 2-7

unlocking, pthread-90, tis-22, E-61, G-69

### Mutex attributes, 2-6

obtaining, pthread-93

### Mutex attributes object

creating, pthread-95, G-60

deleting, pthread-92, G-61

initializing, pthread-95

### Mutexes

thread-safe, 4-2



## N

---

- Names, E-49
  - See Handles
- Nonrecursive mutex, 2-8, pthread-97, E-30, G-63
- Nonreentrant code
  - compilers that generate, 4-1
- Nonreentrant library packages
  - calling, pthread-80, E-56, E-84, G-59
- Nonreentrant software
  - using global lock to avoid, 3-4
- Nonterminating signals, A-4
- Normal mutex, 2-7, pthread-97
- Normal termination of a thread, pthread-54, E-69, E-73, G-45, G-50

## O

---

- One-time initialization routines, 2-13
- OpenVMS Systems, B-1

## P

---

- PASCAL compiler
  - generating reentrant code, 4-1
- Per-thread context, 2-13
  - deleting key, pthread-76
  - generating key value, pthread-74
  - generating key value for, tis-12, E-52, G-57
  - obtaining, pthread-71, tis-11, E-54, G-54
  - setting, pthread-109, tis-35, E-55, G-81
  - uses for, pthread-74, tis-12, E-52, G-57
- Pipelining model, 1-6
- POSIX
  - sigwait service, A-5
- POSIX 1003.1c (pthread) routines, pthread-3
- POSIX 1003.4a (Draft 4) pthread routines, G-1
- Prime number search example, 6-1
- Priority
  - obtaining for thread, E-74, G-52
  - setting for thread, E-79, E-81, G-77, G-79
- Priority attribute, E-24, E-31, G-11, G-19
- Priority inversion
  - avoiding, E-59
  - how to avoid, 3-7
- Process
  - creation, pthread-3
  - states, pthread-3
- Processors
  - causing thread to release control of, pthread-115, E-85, G-84
- Programming
  - synchronization errors, list of, 3-1
- pthread.h, A-2, B-2

- pthread\_debug, pthread-57
- pthread\_exc.h, A-2, B-2
- pthread\_join routine, pthread-72
- pthread\_join32 routine, pthread-72
- pthread\_join64 routine, pthread-72
- pthread\_once\_t data structure, pthread-99, tis-23, G-70

## Q

---

- Queues, E-4
  - creating, F-6
  - creating an attributes object for, F-2
  - deleting, F-7
  - deleting an attributes object for, F-3
  - inserting an element at the end of, F-9, F-12, F-13
  - inserting an element at the front of, F-10, F-14
  - obtaining size of, F-4
  - removing an element from, F-8, F-11
  - setting size of, F-5

## R

---

- Race conditions
  - how to avoid, 3-8
- RAISE exception, 5-3
- Readers locks
  - locking, tis-26, tis-27
  - unlocking, tis-29
- Readers/writer locks, 4-2
  - destroying, tis-30
  - initializing, tis-31
- Readers/writer routines, 4-3
- Recursive mutex, 2-7, pthread-97, E-30, G-63
- Reentrant code
  - See also thread-reentrant code
  - compilers that generate, 4-1
  - necessary for multithreaded program, 1-4
  - nonreentrant routines (avoiding), 1-7
- Reporting errors, 3-9
  - API, 3-9
  - internal, 3-9
- Requeue, E-4
- RERAISE exception, 5-5, 5-8, 5-11
- Routines descriptions
  - DECthreads Library routines, F-1
  - Digital Proprietary Interfaces: cma routines, E-1
  - Digital Proprietary Interfaces: tis routines, tis-3
  - POSIX 1003.1c (pthread) Routines, pthread-3
  - POSIX 1003.4a (Draft 4) pthread routines, G-1
- RR (round-robin) scheduling, 2-4



## S

- Scheduling
  - threads, 3-7
- Scheduling parameters
  - getting for thread, pthread-68
  - setting for thread, pthread-106
- Scheduling parameters attribute, pthread-12
- Scheduling policy
  - getting for thread, pthread-68
  - obtaining for thread, E-75, G-53
  - setting for thread, pthread-106, E-81, G-79
- Scheduling policy attribute, 2-4, pthread-14, pthread-28, E-33, G-21
  - obtaining, E-25, G-12
- Scheduling policy attribute parameters, pthread-26
- Scheduling priority attribute, 2-5
- Scheduling thread, 2-17
- Scope
  - definition of, 3-4
- Sending a signal, tis-25
- Sequence number
  - thread, pthread-70
- Servers, 1-1
- Shared memory, 3-4
- Shared resources
  - access, 4-2
- Shared variables, 3-4
- Signal
  - delivery, pthread-78
  - sending to a thread, pthread-78, tis-25
- Signal mask
  - examining, pthread-111
  - setting, pthread-111
- Signaling a wake-up, pthread-43, pthread-45, tis-8, E-40, E-41, G-36, G-37, G-39
- Signals
  - alternatives to using, A-5
  - asynchronous, A-4, A-6
  - nonterminating, A-4
  - reasons to avoid in a multithreaded program, A-5
  - reported as exceptions, A-6
  - synchronous, A-5
  - terminating, A-4, A-6
  - types of, A-4
- Stack guard area
  - location of, E-21, E-27
- Stack limit
  - checking, E-65
- Stack memory, 3-5
- Stacks, 3-6
  - changing minimum size of, pthread-30, E-35, G-23
  - changing minimum size of guard area, E-27

## Stacks (cont'd)

- obtaining minimum size of, pthread-16, E-26, G-13
- obtaining minimum size of guard area, E-21
- overflow, 3-6
- preventing and detecting overflow, E-21, E-27
- routines for, E-65
- sizing, 3-6

Stacksize attribute, 2-6, pthread-30, E-26, E-35, G-23

- obtaining, pthread-16, G-13

Static memory, 3-5

Storage

- types of, 3-5

Synchronization

- mutex, pthread-84, tis-18, E-57, G-65

Synchronization objects

- atomic queue, E-4
- condition variable, 2-9
- join, 2-13
- mutex, 2-7

Synchronization, asynchronous

- coding for, 3-1

Synchronous signals, A-5

## T

Terminating signals, A-4

Terminating threads

- normal, pthread-64
- premature successful completion, pthread-64
- without returning from start routine, pthread-64

Termination

- waiting for, pthread-72, E-77, G-55

Termination of a thread

- error, pthread-54, E-69, E-72, G-45
- events that cause, pthread-54, E-69, G-45
- normal, pthread-54, E-69, E-73, G-45, G-50
- premature successful completion, E-73, G-50
- without raising an exception, E-72
- without returning from start routine, E-73, G-50

THIS\_CATCH exception, 5-6

Thread

- See also Multithreaded programming, 1-7
- sequence number, pthread-70
- signal to, pthread-78

Thread attributes, 2-4

Thread attributes object

- creating, G-5
- creating and initializing, pthread-18
- deleting, G-7
- destroying, pthread-5

Thread creation

- detached creation attribute, pthread-6, pthread-20, G-8, G-14



## Thread creation (cont'd)

- guardsize attribute, pthread-8, pthread-22, E-21, E-27, G-9, G-16
- inherit scheduling attribute, pthread-10, pthread-24, E-22, E-28, G-10, G-17
- priority attribute, E-24, E-31, G-11, G-19
- scheduling parameters attribute, pthread-12
- scheduling policy attribute, pthread-14, pthread-28, E-25, E-33, G-12, G-21
- scheduling policy parameters attribute, pthread-26
- stacksize attribute, pthread-16, pthread-30, E-26, E-35, G-13, G-23

## Thread-reentrant code

- definition of, 3-3

## Thread-safe code

- condition variables, 4-2
- definition of, 3-3
- mutexes, 4-2

## Thread-specific

- data, 2-13, 4-2

## Threading library

- asynchronous, 3-1

## Threads

- alerting, 2-14
- binding to a CPU, E-68, G-25
- canceling, 2-14, pthread-32, G-26
  - cancelability state, 2-14
  - cancelability type, 2-14
- creating, 2-1, pthread-54, E-69, G-45
- creating detached threads, pthread-6, pthread-20, G-8, G-14
- definition of, 1-1
- delaying execution of, pthread-60, E-48, G-47
- deleting, 2-3, pthread-61, E-71, G-48
- detaching, pthread-6, pthread-20, G-8, G-14
- error termination, pthread-54, E-69, E-72, G-45
- events that cause termination, pthread-54, E-69, G-45
- getting scheduling policy and parameters, pthread-68
- initializing, E-51
- nonreentrant routines (avoiding), 1-7
- normal termination, pthread-54, E-69, E-73, G-45, G-50
- obtaining current priority of, E-74, G-52
- obtaining current scheduling policy of, E-75, G-53
- obtaining handle of, E-76
- obtaining identifier of, pthread-101, tis-32, G-72
- per-thread context of, pthread-74, tis-12, E-52, G-57
- reentrant code necessary, 1-4
- releasing processor, pthread-115, E-85, G-84
- scheduling, 2-17, 3-7
  - inherit scheduling attribute, 2-4

## Threads

### scheduling (cont'd)

- scheduling policy attribute, 2-4
- scheduling priority attribute, 2-5
- setting current cancelability state, pthread-102
- setting current cancelability type, pthread-104
- setting current priority of, E-79, G-77
- setting current scheduling policy and parameters of, pthread-106
- setting current scheduling policy and priority of, E-81, G-79
- signal mask, pthread-111
- starting, 2-1
- states, 1-4
- terminating, 2-1, E-67
  - normal termination, 2-2
- termination, normal, pthread-64
- unlocking global mutex, pthread-114
- waiting for a mutex, tis-20, E-59, G-67
- waiting for another to terminate, 2-2
- waiting for the termination of, pthread-72, E-77, G-55
- waiting on mutex, pthread-86
- waking, pthread-37, pthread-43, pthread-45, tis-3, tis-8, E-36, E-40, E-41, G-32, G-36, G-37, G-39
- yielding processor to another thread, pthread-115, E-85, G-84

## Threads identifier

- comparing, pthread-63, G-49

## Throughput (default) scheduling, 2-4

## Time

- adding interval to current time, pthread-66, E-83, G-51
- obtaining expiration, pthread-66, E-83, G-51

## Timeslice

- definition of, 2-4

## TRY/ENDTRY block

- restriction, B-4

## U

- UNIX, A-2

## UNIX signals

- SIGINT, A-4

## Unlocking

- readers locks, tis-29
- writer lock, tis-42

- Unlocking a global mutex, pthread-114, tis-38, E-84, G-83

- Unlocking a mutex, pthread-90, tis-22, E-61, G-69



## W

- Waiting for condition variable, pthread-47, pthread-49, tis-9, E-42, E-44, G-41, G-43
- Waking a thread, pthread-37, pthread-43, pthread-45, tis-3, tis-8, E-36, E-40, E-41, G-32, G-36, G-37, G-39
- Win32 API, C-1
- WinDbg debugger, D-10
- Windowing system
  - using threads in, 1-1

Windows NT, C-1

Work crew model, 1-5

Work queues

variation of boss/worker model, 1-5

Writer lock

locking, tis-39, tis-40

unlocking, tis-42

## Y

Yielding to another thread, pthread-115, E-85, G-84



